Open Systems Portability Checker

User Guide

Knowledge Software Ltd

November 2007

History

November 07: mcc v5.2, mcl v2.5, mce v2.5
August 05: mcc v5.1, mcl v2.5, mce v2.5
September 99: mcc v5.0, mcl v2.5, mce v2.5
September 97: mcc v4.2, mcl v2.5, mce v2.5
November 96: mcc v4.1, mcl v2.5, mce v2.5
April 96: mcc v4.0, mcl v2.5, mce v2.5
August 95: mcc v3.2, mcl v2.4, mce v2.4
December 94: mcc v3.1, mcl v2.4, mce v2.4
May 94: mcc v3.0, mcl v2.4, mce v2.4
July 93: mcc v2.3b, mcl v2.3, mce v2.3
December 92: mcc v2.3a, mcl v2.3, mce v2.3
July 92: mcc v2.3, mcl v2.3, mce v2.3
February 92: mcc v2.2, mcl v2.2, mce v2.2
September 91: mcc v2.1, mcl v2.1, mci v2.1
December 90: mcc v2.0, mcl v2.0, mci v2.0
August 90: mcc v1.0, mcl v1.0, mci v1.0

Support

Knowledge Software Ltd provides telephone and mail support for those users who have purchased their systems from Knowledge Software Ltd. All other users of this system must contact their supplier for support. Knowledge Software Ltd does not have the resources to support users who have purchased their software from other vendors.

Disclaimer

TradeMarks

Knowledge Software Ltd. Farnborough, Hants GU14 9RZ, England.

e-mail: OSPC@knosof.co.uk   Tel: +44 (0) 1252-520667

Web: www.knosof.co.uk

TABLE OF CONTENTS

Chapter 1

Introduction

Open Systems were designed to enable the portability of applications across platforms. To achieve this portability both the platforms and the applications must conform to the standards. The job of ensuring that a platform conforms to standards is the responsibility of the hardware vendor.

There is no point in having a platform that conforms to standards if the application does not itself conform to them. **OSPC** is a set of interrelated tools that check software, written in the C language, for conformance to the POSIX, XPG, X-windows and ANSI C standards (amongst others). **OSPC** is capable of operating at all phases of the creation and execution of a program; compile, link and runtime. This guide deals with these aspects of standards checking that do not require the application to be executed, that is, checks that are performed statically.

This User Guide describes the process of creating applications software that conforms to Open Systems standards. It also gives a brief overview of each tool in turn, fuller details can be found in the User Reference manual. It also describes the tools' common user interface and how they interact with each other.

The use of standards offer the software developer the opportunity for a significant reduction in cost and effort when porting applications to different platforms. In practice there are two main reasons why applications fail to conform to the requirements of POSIX and other standards. The immediate problem is one of know-how and old habits. Once these are overcome, problems are caused by human oversight and error.

Because of the broad range of services offered it can take some time for developers to think POSIX. Old, Unix, programmer habits and know-how are easily transferred to a POSIX development environment. Programmers cannot be expected to be familiar with all the intricacies of POSIX and how it differs from what they are familiar with. Speaking Unix with a POSIX accent will not solve portability problems, particularly to proprietary platforms that support POSIX. It is necessary to speak POSIX as a native language and if using Unix perhaps with a Unix accent. Training can go someway towards ensuring a smoother transition to a POSIX only environment.

Experience over 40 years of software development has shown that it is impossible to produce any significant applications that do not contain bugs. The same principle holds true for writing POSIX conforming applications. Mistakes will be made.

So some means of independently and accurately checking conformance could uncover most of these problems and save a considerable amount of time and money later. Studies have shown that the later a problem is discovered the more expensive it is to fix. Thus the obvious time to find these non-conforming constructs is before the release of the software.

From the marketing perspective Open Systems are being demanded by users. Use of an independent verification tool to check conformance will add weight to any claims of conformance to Open Systems Standards made by software vendors. From the users perspective, demanding such verification is a useful means of ensuring vendor compliance with any Open Systems agreements that they may have.

For those developers considering a move to POSIX, information provided by a checking tool can be used to provide an estimate of porting costs for existing applications. By providing hard information on likely problems, time/cost estimates for porting an application are likely to be much more accurate than uninformed estimates.

1.1 Background

The **OSPC** has been derived from the Model Implementation C Checker. A Model Implementation is a compiler, linker, library and runtime system that follows the exact letter of a language standard. Thus it can be used to check application programs for strict conformance to the C standard. There are also other uses. For instance, Model Implementations have been used to test validation suites which are to be used to check other compilers for conformance to the Standard (both NIST and BSI have used the C Model Implementation for this purpose).

Model Implementations have been produced for Pascal and Ada. In March 1989 the British Standards Institution signed an agreement with Knowledge Software to produce one for C.

1.1.1 How closely does the OSPC follow Standards?

The design aim of a Model Implementation is to follow the exact letter of a language standard. The ANSI C Standard allows compilers leeway in that many features are left undefined, implementation defined or unspecified. Thus it is possible for a program that compiles and runs with one compiler to fail to compile or give different results at runtime.

The source of the **OSPC** has been cross referenced to the C standard by page and line number. There is also a suite of test programs that cause all statements in the **OSPC** to be executed. Of course the **OSPC** also passes both the NIST (from Perennial Inc) and BSI (from PlumHall Inc) C validation suites. All in all a very large amount of effort went into showing the correctness of the software from which the **OSPC** was derived. Users can thus have a high degree of confidence that the results it gives are correct.

1.1.2 Design aims

The design aims of the **OSPC** were as follows:

- Flag all non-strictly conforming uses of C constructs. This includes syntax, constraint, undefined, implementation defined, unspecified behaviour and any minimum limits that are exceeded. Checks to be performed at all stages of the software development cycle.

- Be able to check against a wide range of API specifications.

- Be user configurable. There are very few ‘hard coded’ internal values. Nearly everything being read from configuration files, which can be changed by the user.

- Informally prove that the **OSPC** correctly processes the language as described in the C Standard.

- Analyse the application quickly. Being derived from a Model Implementation should not be an excuse for low compilation rates.

- All implementation defined and undefined behaviour to be user selectable.

- Software to be portable across a range of architectures. This means that the **OSPC** can be ported quickly to a wide range of platforms.

1.2 Why use the Open Systems Portability Checker?

So what does the **OSPC** detect that a development compiler, or lint would overlook? The C standard allows compiler writers considerable leeway in the handling of many constructs. The reason the C standards committee (X3J11) permitted this leeway was that many compilers already silently handled these constructs one way or another and the desire was to codify common existing practice. Thus although a compiler may pass a C validation suite it may still leave, and is perfectly entitled to leave, many constructs unflagged. By specifying the behaviour for many combinations of constructs to be undefined implementors have been given freedom to decide what to do. This freedom means that C programs can behave differently with different C compilers, even on the same machine. There are no requirements on compilers to flag occurrences of these undefined constructs. The **OSPC** was designed to detect all non-strictly conforming behaviour in C programs. It detects and flags all constraint errors, implementation defined, undefined, unspecified behaviour and exceeding minimum limits; at compile, link and runtime. The following points highlight the differences:

OSPC

- Check strict-conformance to the C standard. Pick up standard requirements missed by the development compiler.

- Check use of system service interface. Perform symbolic checks on parameters and return values to ensure that the defined interface is being used.

- Can be tailored to mimic a variety of platforms and the components from which they are created.

- Detect: Undefined, Implementation defined and unspecified behavior as well as constraint errors.

- Cross translation unit checks to verify interface correctness.

- Support for multiple architectures.

Development compiler or lint:

- May not be ANSI conforming.
- Likely to be silent on constructs the exhibit undefined, implementation defined or unspecified behaviours.
- May 'hide' the implementation defined features from its users.
- Unlikely to check across translation units.
- Only likely to support a single machine architecture.

1.2.1 Support for multiple architectures

The **OSPC** is not tied to any computer architecture. It can be configured to emulate various architectures. This means that it can be used to check software for portability to other systems. The user can configure the static checking component tools of the **OSPC** to match:

- The development compiler
- A variety of cpus
- The commonly used operating systems
- Combinations of various standards

1.3 How to use this guide

This guide is aimed at those users who want a quick introduction on how to use the **OSPC** for checking their software. More detailed information can be found in the User Reference Manual.

Users of the **OSPC** would not normally need an introduction to the C language or the edit/compile/run cycle of program development and none is given.

Most people try the software first. If they cannot get it to work they read the manual. The mode of operation of the **OSPC** will be familiar to all developers and so this approach will work to some degree. Also the on-line help enables many questions to be answered. However, The **OSPC** is a sophisticated piece of software. To get the best from it some background knowledge is required. It is not recommended that you read this guide from cover to cover, before doing anything (other than falling asleep). Rather, it is suggested that you try the software while reading its documentation.

The chapter entitled Getting Started (it follows this one) is a good place to begin.

1.4 Contents of Guide

The following gives a more detailed look at what the User’s Guide contains:

1.5 Related documents

Installation guide, User Reference manual

ISO C Standard. ISO/IEC 9899:1990 (ANSI C Standard. X3.159-1989)

POSIX.1 ISO/IEC 9945-1 (system API)

Unix System V Release 4 Programmer’s Guide: ANSI C and Programming Support Tools

C Language Interfaces, AT&T Data Systems Group, 1989. ISBN 0-13-109661-3

Go Solo, go solo with the single unix specification. ISBN 0-13-439381-3

1.6 Conventions

References to the Standard, when a language is being discussed, should be taken to mean ISO/IEC 9899:1990.

The typographical conventions used follow those given in the POSIX standards.

Type of entry	Example
C-Language Data Type	*short int*
C-Language Error Number	[EINVAL]
C-Language Function	*printf()*
C-Language Argument	*stream*
C-Language Global External	*errno*
C-Language Header	`<stdio.h>`
C-Language Keyword	`#undef`
Constants	MAX_UCHAR
Environment Variables	**MCEDITOR**
Example Input	`mcc myprog`
Example Output	**Hello world!**
File Name	`/usr/include`
Special Character	`<new-line>`
Utility Name	**mcc**
Utility Option	`-CFG`

Type of entry	Example
Parameter	**[**<*platform type*>**]**

1.7 Reporting problems

Problems can be reported via electronic of paper mail. A bug report form can be found in the distributed software package in `doc/prob.txt`.

Our electronic address is:

support@knosof.co.uk

Up todate information can also be found on our web site at www.knosof.co.uk

Suggestions for improvement are also welcome.

Note: These tools check the requirements given in standards documents. If you are unhappy with these requirements you should address your complaints to the relevant committee. Don't shoot the messenger.

Chapter 2

Getting started

2.1 Introduction

This chapter will take you through the process of checking several applications for conformance to various standards, using **OSPC**. We will take existing programs that contain a number of non conforming and non portable constructs and go through the process of detecting and removing them. Because of the wide range of services provided by different platforms the range of warnings generated by **OSPC** can be very large. The use of platform profiles is the key to controlling this diversity. The description on how to use the tools will therefore go hand in hand with the concept of platform profiles.

We assume that the **OSPC** has been correctly installed on your system. This manual deals with the static portion of **OSPC**. That is, those checks that can be performed without having to run the application. Another set of manuals deals with the checking of the runtime characteristics of an application. See below for information on checking for correct installation.

We will go through the steps:

- Detecting and removing constructs flagged in the source code.
- Detecting and removing constructs flagged at cross unit checking time.
- Showing how platform profiles affect the behaviour of the **OSPC** tools.
- Showing how **OSPC** can be integrated into an existing development environment.

2.2 Checking the installation

The following procedure will check that the **OSPC** has been correctly installed.

Issue the following command (at the Unix shell prompt):

```
$ mcc
```

this should cause the **mcc** help screen to appear, and issuing the command:

```
$ mcl
```

should cause the **mcl** help text to appear.

In each case the help text should include half a dozen, or more options. If only a two line summary of the command line syntax appears, then go back to the installation notes.

OSPC is licensed on a per seat basis. Before running each tool checks to ensure that no more than the permitted number of users are already running. If the maximum number of users has been reached a message will be displayed and the tool aborted. The user will then have to wait for a slot to become free.

Included with the software is a directory of example programs. The directory is called `example` and contains various subdirectories. Before going through this example it is recommended that a temporary working directory be made and the contents of `checker/example` copied into it.

2.3 Example 1

A directory listing should reveal the files: `Makefile, main.c, util.c, main.via` and `.mccrc`.

```
/*
    main.c, 16 Jan 91   Main unit of program for generating
                        prime numbers.  Uses Euclid’s method.
   Copyright (c) 1991, Knowledge Software Ltd.
*/
#include <stdio.h>

#define MAX_PRIMES 300
long primes[MAX_PRIMES];
long Calculate_primes();
void printf_results(long);

void main(void)
{
long num_primes_found;
printf(“Calculate the primes that occur in the first %d numbers\n”,
                                                     MAX_PRIMES);
num_primes_found = Calculate_primes(MAX_PRIMES);
printf_results(num_primes_found);
}


/*
     utils.c, 16 Jan 91 Utility routines called from main.c
                        to calculate prime numbers
*/
extern int primes[];

int Calculate_primes(int max_primes)
{
unsigned int primes_found_m1 = 1,
             num_to_try = 5;
/*
   Set up the first two
*/
primes[0] = 2;
primes[1] = 3;
while (num_to_try < max_primes)
  {
   int loop_index;
   for (loop_index = 0; loop_index <= primes_found_m1;
                                            loop_index++)
      if ((num_to_try % primes[loop_index]) == 0)
/*
```

```
   if the remainder is zero then this number is not prime
*/
         {
         num_to_try += 2;
         break;
         }

   if (loop_index > primes_found_m1)
      {
      /*  found one */
      primes_found_m1++;
      primes[primes_found_m1] = num_to_try;
/*
   Only odd numbers (after 2) are prime.
*/
      num_to_try += 2;
      }
   }
/*
   return how many we found
*/
return primes_found_m1+1;
}

void printf_results(int primes_found)
{
/* print out the prime numbers that have been found */
signed char loop_index;

for (loop_index = 0; loop_index <= primes_found; loop_index++)
   printf("%d is prime\n", primes[loop_index]);
}
```

In any large development project **make** would normally be used. But we shall start off describing how each component, of **OSPC**, operates before fully integrating them. The first task is to compile `main.c` and `util.c`. To do this type:

mcc main

(the .c need not be given) and:

mcc util

More than one file can be given on the command line:

mcc main util

would have the same effect.

Several things will have happened. Various warnings appear on the screen and two new files will have been created, `main.kic` and `util.kic`. Since only warnings occurred it would be possible to go on to check the units for interface consistency. However, it is probably wiser to remove of the warnings generated by **mcc** first.

2.3.1 Removing the mcc problems

If there are only a few messages it is usually easy to remember which line they occurred on. Larger numbers of messages can be dealt with by creating a log file:

mcc main -log main.log

in this case the option `-LOG+` would have had the same effect (since the name chosen was the same as the source file being checked). To create listing file (which will include all lines in the source, not just those flagged) type:

mcc main -l+

or, of course, by using the Unix redirection facility:

mcc main > main.output

Looking at the generated warnings we see that there are two warnings concerning `main.c`:

1 The function *printf_results* is the same, within six characters as the function *printf*. The solution here is simple. Rename *printf_results* to *print_results*.

2 The second warning concerns the function *main*. Since nothing is passed to this function and nothing returned the user has defined it appropriately. However, *main* is a special function. The Standard specifies that it can only be defined in one of two ways. In this case the appropriate one is *int main(void)* (since main does in fact return a value to its caller).

Compiling `utils.c` also gives the warning about *printf_results* and *printf*.

To find out where about in the standard these restriction are described, turn on standard references (using the `-REF+` option).

mcc main -REF+

In the case of the first problem you will be told that section 5.1.2.2 of the ISO C Standard is the place to look.

2.4 Using mcl

In order to perform cross unit checking we need to link the two translated units together. The simplest method is to type:

mcl main utils -Lib+

or (if the file `main.via` contains the lines `main`, `utils` and `-Lib+`):

mcl -VIA main.via

Once it is running **mcl** will report on its progress, generate some warnings and produce a file called `main.klc` (the first filename on the command line is used as the basis of the output file, unless overriden with the `-Output` option).

The option `-Lib+` tells **mcl** to use the default library when checking the unit interfaces. It is also possible to specify the pathname of a library .klc file at this point. However, this operation is so common that an option was created to carry it out. Our program contains a call to the *printf* library function (which has not been defined in the calling translation unit). It is contained in the C library stdio.

2.4.1 Removing the mcl warnings

mcl operates on a translated version of the source code. Its main checking role involves comparing externals of the same name, declared in different files, for compatibility. Its output takes the form of a name, followed by the types involved and a message describing the problem and the names of the files containing declarations/definitions of those names. There is no line number information (this information could, in theory, be made available, but it would significantly increase the size of the .kic files and the problem of multiple declarations of the same name in the same file would have to be addressed).

In the case of this example there are four warnings generated by **mcl**.

The first is caused because an object, *primes*, has been declared with different types in the two source files.

The other warnings are telling us that a function declaration in one unit is not compatible with a function definition in another unit (the function types are not compatible).

The programmer obviously forgot what the parameter and return types were when coding up these units. In one file a function was declared using *int*, but *long* was used for the definitions in the other file.

The solution is to make the types of the declaration and definition agree, in both files. Here it would be possible to use the types *int* or *long* for the parameters and return types. The only requirement is that the same choice has to be made in both files.

For arguments sake, lets edit `main.c` and `utils.c` to use *int*s throughout. Having made this change we now have to recompile the two units, using **mcc**. This time, when they are processed using **mcl** there are no warnings generated.

2.5 Platform profiles

The secret of getting the best out of the **OSPC** tool set lies in making full use of platform profiles. Platform profiles offer two benefits:

1. They provide a convenient means of organising the attributes of different platforms, so users need not concern themselves with details and,

2. They provide a method of reducing the number of warnings generated when a specific translation unit is processed (in its raw mode **OSPC** can flag a large number of potential portability problems).

Putting in the effort to make software portable to all platforms can be wasted effort. Normally the **OSPC** complains about everything that could cause portability problems. However complaining about all portability problems can result in a large quantity of output. Platform profiles provide a simple way of reducing the quantity of output, thus highlighting the essential information. The user selects the source and target platforms before running the tools. The **OSPC** uses this information to select which warnings are relevant for the portation being undertaken. The reduction in the number of warnings generated is brought about because different platforms often share many characteristics with each other. Thus, knowing that the software being processed already runs on one platform (the source) and knowing the identity of the target platform, it is possible to filter out warnings about those constructs that are common to both. The idea being that if the construct already works on the source then it is highly likely that it will also work on the target platform.

Profiles come in two types - platform and component profiles (they may be represented in text or binary files). Platform profiles are in fact built up from component profiles. The component profiles contain, as the name suggests, information about the components of a platform (ie cpu, compiler, OS, standards, etc). In the following examples we shall start off by using existing platform profiles and seeing what effect different profiles have on the warnings generated. We shall then go onto to show how platform profiles can be extended to include additional standards; for instance a sun4 profile modified by a X11 subprofile instead of Sun View. A full description of the information held in platform profiles and its format can be found in the User Reference Manual.

2.6 Example 2

In the first example the affects of using platform profiles was skipped. Here we shall consider an example where changes to the platform profiles can have a dramatic effect on the warnings generated.

```
/*
    libfun.c, 25 May 91
    Copyright (c) 1991, Knowledge Software Ltd.
*/
#include <stdio.h>

char gac[20];
char *gpc;
int *gpi;

int total;  /* common enough name */
#define COUNT_MAX 33;  /* looks ok */

char *p1, *p2;

extern int loop_counter_a,
           loop_counter_b; /* first 13 characters the same */

void func(void)
{

bcopy(p1, p2, len);  /* platform specific memory copy routine */

gpi = (int *)gac; /* potentially different alignments */

gpc = (char *)&loop_counter_a; /* byte sex dependency */

*gpc = '\a'; /* making use of ISO C feature */
```

```
total >>= 1;

/* ... */
```

The above program contains several constructs that may, or may nor give rise to portability problems. An cursory examination of the source would suggest that this software was written for a machine with few alignment restrictions and a C compiler that was closer to the ISO C standard than K&R. If the target platform has the same attributes there should be few porting problems.

However, if the target platform has strict alignment requirements on the addresses of objects, a dereference of the address contained in *gpi* (after it had been assigned a value) could well result in the cpu raising an alignment fault. The problems cause by the subsequent assignment are likely to be more subtle. The address of an object of type *int* is being taken. Now if both the source and target processors have the same byte sex there will be no problems. However, if they have opposite byte sex any access to the storage by dereferencing *gpc* will result in a different part of the *int* object being used than expected. In the final assignment use is being made of an ISO C specific feature. The character constant *'\a'* is replaced by the alert character (beep) under ISO. On pre-ISO compilers the escape sequence has no special meaning and is likely to have the same effect as the character constant *'a'*.

This code is also relying on characteristics of the host library (the call to *bcopy*), linker (number of significant characters in external identifiers) and cpu (is arithmetic right shift signed or unsigned?). Use is also being made of identifiers that are reserved for future use by the C standard and the POSIX.1 standard (use the **mcc** option `-REF` to find out where the standards reserve these names).

To see what effects the source and target platform profile have on the warnings generated for this file try the following:

mcc libfunc -src unknown -tgt unknown

mcc libfunc -src unknown -tgt cabstract

mcc libfunc -src cabstract -tgt posabstract

mcc libfunc -src unknown -tgt sun4

mcc libfunc -src sun4 -tgt 88k

mcc libfunc -src dos -tgt sun4

In practice users are unlikely to use such a wide range of source/target platforms. A local configuration file (`.mccrc`) can be created to hold frequently used command line options.

2.7 Example 3

Change to the directory `example/example1`. The file `getpswd.c` contains a function whose job it is to obtain a password from the user, typed on *STDIN*. To stop a casual bystander seeing the characters typed, the function switches off echo while the password is being input and then re-enables it.

The function does not conform to the published POSIX and C API’s in a number of places.

```
                   /* getpswd.c, 26 Mar 96 */

/*
 * Taken from
 * POSIX programmers guide by Donald Lewine, ISBN 0-937175-73-0
 */

#include <errno.h>
#include <termios.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int getpswd(char *buff, unsigned size)
{
struct termios attr;
int n;

if (printf(“Password: ”) == -1)
   return -1;
if (fflush(stdout) == -1)
   return -1;

/*
 * Get attributes and turn off echo
 */
if (tcgetattr(STDIN_FILENO, &attr) != 0)
   return -1;

attr.c_lflag &= ~(0x10);

if (tcsetattr(STDIN_FILENO, 2, &attr) != 0)
   return -1;

/*
 * Read the password type on stdin
 */
errno=0;
n=read(STDIN_FILENO, buff, size);
if (errno != 0)
   return -1;
/*
 * EBADF not a valid file descriptor
 * EAGAIN The O_NONBLOCK flag is set and the process cannot be
                                                       delayed
 * EINTR operation was interrupted by a signal
 * EIO background job is attempting to read from its controlling
                                                        terminal
 */

/*
 * Now reenable the echo
 */
attr.c_lflag |= 0x10;

if (tcsetattr(STDIN_FILENO, 0, &attr) != 0)
   return -1;

57:return n;
58:}
```

Line 19. “The printf function returns the number of characters transmitted, or a negative value if an output error occurred.” ISO C Clause 7.9.6.3. As written the code is not testing the negative property, but for a particular instance of a negative value. The correct test is for less than zero.

Line 21. “The fflush function returns EOF if a write error occurs, otherwise zero.” ISO C Clause 7.9.5.2. The code assumes a particular value for the *EOF* macro, *-1*. There is no requirement that the *EOF* macro have this value on all implementations.

Line 30. “Values of the *c_lflag* field, shown in Table 7-4, describe the control functions and are composed of the bitwise inclusive OR of the masks shown ...” ISO POSIX Clause 7.1.2.5. The code assumes that the *ECHO* macro has the value *0x10*. There is no such requirement in the API.

Line 32. ISO POSIX Clause 7.2.1.2 defines actions when particular symbolic values are passed as the second parameter. The code assumes that the macro *TCSAFLUSH* has the value *2*.

Line 40. There is no requirement in the POSIX API that return codes be checked. When running in API mode only no check is made that errno is tested for.

Line 52. Re line 30. The macro *ECHO* should be used.

Line 54. Re line 32. Except in this case the macro *TCSANOW* is intended.

If the source code is modified to take into account the API interface violations described above we get the code shown below.

```
                   /* getpswd.c, 26 Mar 96 */
/*
 * Taken from
 * POSIX programmers guide by Donald Lewine, ISBN 0-937175-73-0
 */

#include <errno.h>
#include <termios.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int getpswd(char *buff, unsigned size)
{
struct termios attr;
int n;

if (printf(Password: “)  0)
  return -1;
if (fflush(stdout) == EOF)
  return -1;

/*
 * Get attributes and turn off echo
 */
if (tcgetattr(STDIN_FILENO, &attr) != 0)
```

```
  return -1;

attr.c_lflag &= ~(ECHO);

if (tcsetattr(STDIN_FILENO, TCSAFLUSH, &attr) != 0)
  return -1;

/*
 * Read the password type on stdin
 */
errno=0;
n=read(STDIN_FILENO, buff, size);
if (errno != 0)
   return -1;
/*
 * EBADF not a valid file descriptor
 * EAGAIN The O_NONBLOCK flag is set and the process cannot be
                                                      delayed
 * EINTR operation was interrupted by a signal
 * EIO background job is attempting to read from its controlling
                                                        terminal
 */

/*
 * Now reenable the echo
 */
attr.c_lflag |= ECHO;

if (tcsetattr(STDIN_FILENO, TCSANOW, &attr) != 0)
   return -1;

return n;
}
```

2.8 Integrating OSPC into the existing environment

The components of **OSPC** can be run as stand alone tools. However, most applications packages contain a large volume of source spread over many directories. Also this source can often be built in one of several ways (usually to support different configurations). To use **OSPC** in this environment means that it is necessary to be able to make use of existing development tools, in particular make files.

2.8.1 Example 1 revisited

Change back to the working directory that contains this example and type **make**. The files will be compiled and linked using the host compiler and linker.

Remove the two files `main.o` and `util.o`, then type:

make CC=ccc

ccc is a script that causes both the development compiler and **mcc** to invoked on the source files. If a link is needed **mcl** will be invoked.

The command line option `CC=ccc` overrides the value given to `CC` (an environment variable) inside the make file. If the line `CC=cc`, in the make file, was changed to `CC=ccc`, this command line option would not be necessary.

If **mcc** is not invoked on every file that is compiled, by the host compiler, there are several possibilities:

1. Within the body of the make file any occurrence of **cc** should be changed to `$(CC)`. The line

   ```
   CC = cc
   ```

   should appear somewhere within the file. It is possible that this line does not already exist in the make file. In this case it can simply be added near the start of the file, with the other macro definitions.

2. The lines:

   ```
   .c.o :
   	$(CC) $(CFLAGS) $<
   ```

   should be added, if they don't already exist, to the make file (note that the second line starts with a tab character, not multiple space characters). They describe the default rules for creating a .o file from a .c file. Note that on some platforms the default rule for invoking **cc** also define some macros via the `-D` option.

3. If use is made of **ar** to build libraries the line:

   ```
   AR = ar
   ```

   has to be changed to:

   ```
   AR = arr
   ```

   It is possible that this line does not already exist in the make file. In this case it can simply be added near the start of the file, with the other macro definitions. Similarly within the body of the make file any occurrence of **ar** should be changed to `$(AR)`. This will cause libraries of .kic files to be created.

By having a make file that explicitly contains the rules for creating a .o file from a .c file the user at least knows what is going on. Sometimes the inbuilt default rules contain non-portable options to **cc**, which **ccc** cannot understand. By providing an explicit default rule this problem is avoided. In some cases users may not want to change the value of the `CC` macro to be **ccc**. In this case **make** can be invoked with the command line option, as was done above.

2.9 Using platform profiles

Platform profiles are not designed to provide the definitive answer in terms of how software should be targeted to a given platform. The information given in a platform profile provides the core information. Users may want to modify, from time to time, some of the settings. The .rc file capability is provided to allow this modification to be carried out on a project basis and command line options allow option settings to be changed on a per instance basis. The chapter on the user interface and configuration describes how local configuration files work.

Options are read from many different files and it can sometimes be difficult to work out where a particular option is being modified. Two options are provided to aid debugging of configuration files; `-TRACE options` and `-TRACE profiles`. Specifying either of these options causes the contents of the configuration file to be sent to `stdout`, as it is being read in. Option tracing switches output on for all options that are read from files, whist profile tracing only displays the options defined in profiles.

All of these configuration files can sometimes make it difficult to work out the option values finally used by **OSPC**. Help is at hand in the form of the help text. Alongside the description of each command is the current setting of that option.

Chapter 3

Creating portable software

3.1 Introduction

This chapter provides an overview of the process of creating portable applications software. It discusses the ideas behind portable software and looks at the management and technical issues involved. It also gives an overview of the tools provided within **OSPC** for detecting non-conforming constructs in C programs.

3.2 What does portability mean?

Portability means different things to different people. Standards define the term 'strictly conforming'. Software that strictly conforms to the requirements of the C and POSIX standards is maximally portable with regard to those standards. But is such software maximally portable with respect to existing hardware platforms? Probably not. The reason is that there are many platforms that do not conform to the POSIX standard, either through intent (or lack of intent) or oversight. To be maximally portable an application should not only keep within defined standards, but also only use those features that are known to be widely available.

Do developers want to create maximally portable software? It sounds like a good idea. What are the drawbacks? Mainly time and money. Creating software that is likely to run on any reasonably compliant platform will take time and effort. Thus developers need to take a look at the costs of creating maximally portable software verses the likely savings to be achieved. It may well turn out to be more cost effective to create software that is portable to a range of platforms, rather than all platforms.

The tools described here are based on the concept of platform profiles. Platform profiles were introduced to allow developers to specify the platform(s) they would like portability to. By knowing the target platform it is possible to filter out warnings about those constructs that are not of interest for that platform. Knowing the source platform (where the software is known to run correctly, or at least assumed to run correctly) it is possible to reduce further the number of 'uninteresting' warnings generated. Platform profiles also act as a method for encapsulating the configuration information about various platforms. Once a platform profile has been created users need no longer concern themselves with the inner details of particular platforms.

Creating a strictly conforming C translation unit (what the C Standard calls a stand alone text file that can be compiled) will probably be tougher than you think. The C standard contains a large number of constructs whose use results in undefined or implementation defined behaviour. Existing compilers silently process these constructs. Most of the time this processing results in programs that behave as expected. However, every now and then unexpected behaviour occurs; resulting in strange bugs that can take a long time to track down.

C is also a more complex language than most people think. There are many subtleties in its definition. The ANSI C (the standard was written by the ANSI committee X3J11 and was subsequently adopted as an ISO standard, ISO/IEC 9899:1992) committees' decision not to break existing code has led to many twists and turns. This complexity is not visible to most users. The idiosyncracies of the compiler in use are probably well known and developers have adjusted to its view of the world. The **OSPC** is different. It is very fussy. It doesn't just complain about what it is required to complain about. It will complain about the complete set of constructs that the C Standard allows to be complained about, plus those contained in any other standards that are referenced.

3.2.1 Don't compilers check?

The development compiler is only likely to check for constraint and syntax errors, since it is these constructs that a conforming implementation is required to detect (and must detect in order for a compiler to validate). Also the compiler's job is to check the language as defined in the C standard, it is not interested in checking the requirements contained in other standards.

One of the principles behind the drafting of the C standard was that existing code should not be broken by wording in the standard. This meant that in many cases the behaviour was left undefined, implementation defined or unspecified. By not specifying what had to be done, compiler implementors were free to make their own decisions. Thus preserving the correctness of existing, old code. So in general, compilers are silent on those constructs whose behaviour may vary across implementations. This freedom means that C programs can behave differently with different ISO validated C compilers, even on the same machine. There are no requirements on compilers to flag occurrences of these non constraint/syntax errors.

The C standard committee also recognised that compiler vendors would have to rely on existing tools to link separately compiled units together. Since existing linkers were unlikely to check for external variables and functions for inconsistencies between modules it was felt that the C standard should not mandate such checks.

Runtime checking is not considered to be in the spirit of C programming. Thus compilers do not generate code to check that pointers are within bounds, that the correct number of parameters are passed or check that any of the runtime conditions are violated.

3.3 What to check

Having shown the benefits of conforming to POSIX and that the best way of achieving this is to use some form of checking tool we now have to investigate what constructs ought to be flagged and why. There are two main sources of information on constructs that ought to be checked to achieve applications portability:

1 The text of standards documents. Here we are interested in applications written in the C language. So the relevant standards are the C language standard and the C language bindings provided by the POSIX.1 (ISO 9945-1) standard (the other POSIX documents are still drafts and have yet to achieve formal standards status).

2 Practical experience. The sources for this information tend to be first hand experiences and conversations with developers on problems that they have encountered. Books on software portability are starting to appear. But on the whole these tend to give general guidelines rather than cover specific cases. One problem with specific cases is that they go out of date. As compilers and O/S's evolve problems disappear and new ones appear.

The core of the POSIX checker is driven by the requirements given in the C and POSIX.1 standards. Messages are categorised in the same manner as the standards documents. Also any construct which is not strictly conforming is flagged. Provided with these core checking abilities the user can then provide configuration information (done via source and target profiles, discussed later) to switch off any messages that are not of interest.

Thus no justification, other than appearing in a standards document, is given for flagging these core constructs. Those developers familiar with the standards process will know that the contents of standards are sometimes driven by immediate political needs rather than technical merit. Attempting to weed out the political from the technical issues was not considered to be worthwhile. Matters are greatly simplified (from our point of view) by simply handling all constructs.

The necessity for checks based on practical experience occurs because we live in an imperfect world. Operating systems and compilers do not fully conform to standards and contain bugs. Sometimes these bugs are actually features, they are there for compatibility with previous versions of the software. The justification for flagging these constructs goes along the lines of "this construct is not supported/behaves differently on the xyz platform". From this observation we draw the conclusion that truly portable applications have to be written using a subset of the facilities and services described in standards documents.

3.3.1 Standards conformance

The POSIX and C standards define two types of conformance, 1) implementation conformance (ie the OS) and 2) application (or source code) conformance. This tool set checks the latter.

Application conformance is broken down into various categories. The classification of these categories varies slightly between the two standards.

3.3.2 POSIX specifics

POSIX itself is not specific to the C language. However, it does have a C binding (ISO 9945-1, currently being revised into a language independent and language dependent standards). This binding specifies an interface to the environment, but surprisingly there are no requirements in POSIX.1 for the C source code to conform to the C standard. However, from the portability perspective any software that conforms to the C standard should be portable across C compilers running in a POSIX environment. So here we will be considering the POSIX and C standards as one.

A strictly conforming POSIX.1 program does not rely on any construct whose behaviour is not fully defined, thus it has the greatest portability. A conforming POSIX.1 application may only use facilities described in the standard, or other accredited standards. However, since the behaviour of some of those facilities may vary across implementations such an application may need to be modified to run on different platforms.

The POSIX.1 standard also defines <National Body> conforming applications and conforming applications using extensions. It is expected that applications conforming to these standards will have weaker portability criteria and are not considered further here.

The POSIX standards are very new. A consequence of this, is that the moment there are constructs for which it is uncertain (at least to the author) which category of behaviour they cause. As time passes the user community will iron out the problems caused by inconsistencies or missing wording in these documents.

3.3.3 C specifics

The C standard defines terms for a strictly conforming and conforming programs. From the perspective of applications software the C standard defines a language. This language has a particular view of the world and other standards must use this when defining a C binding to a particular set of services.

The C standard is all encompassing in that all constructs can be categorised. Over the last few years there has been a considerable debate about the status of various C constructs. This has resulted a feeling that any remaining poorly defined constructs are likely to be obscure. There is an active program of documenting answers to interpretation questions raised by users of the C standard.

3.3.4 C/POSIX.1 differences

The major difference between the POSIX.1 and C standards occurs at runtime. POSIX specifies a much larger set of support functions. Basically it provides an interface to the host operating system, whereas the C standard provides library functions independent of the host OS. The topic of runtime conformance checking is dealt with in the runtime documentation.

The rules and regulations governing the creation of a runable program are specified to be those given in the relevant language binding. In the case of the C language binding some of the minimum limits given in the C standard are increased, i.e., number of characters considered significant in an external identifier.

3.4 Components of the OSPC

Creating an executable program from C source code requires four components: 1) compiler, 2) linker, 3) library and 4) runtime system.

3.4.1 The Compiler

It was recognised at an early stage that most existing C programs are a long way from being strictly conforming. The user interface of the **OSPC** was designed to smooth the transition from K&R and common usage C to conforming ISO C. Not only is it possible to tailor the severity of every error message but implementation defined features are user selectable. This tailoring enables users to convert their source code in an incremental fashion. Thus the work load can be spread over a period of time. It is also possible to achieve results quickly, rather than having to wait until all the work is complete.

The compiler has no hardwired internal limits and will handle any large program, given sufficient memory.

3.4.2 The Cross unit checker

The **OSPC** cross unit checker (linker) was tailor written for linking C programs. Most linkers perform very little checking across translation units. They are usually restricted to complaining about missing symbols.

The **OSPC** cross unit checker performs full type checking across C translation units, i.e., it checks that the same identifier is declared with compatible types in every file in which it is referenced.

3.4.3 The runtime interface checker

This portion of **OSPC** is provided in the dynamic checking package, separate from the static checking portion along with the runtime system. The **OSPC** supplies the functionality required by the POSIX and C standards by providing a set of routines that interface to the host libraries. Following the design aims of the previous phases these interface routines also give warnings on the use of any undefined, implementation defined or unspecified behaviour that occurs. The interface routines are called prior to the call to the actual system service routine.

3.4.4 The Runtime system

This portion of **OSPC** is also provided with the runtime interface checker and is packaged separately from the static checking portion. The **OSPC** runtime system executes the code generated by the compiler and performs checks on the correctness of operations. For instance, checking that memory accesses via pointers do not result in stray memory references; warning when casts result in loss of information and checking that function parameters are correctly accessed.

3.5 When constructs are flagged

Ideally it would be possible to flag every construct that should be flagged, by doing a static analysis of the application source. In practice this is not possible. In order to detect all possible occurrences of non-conforming standards constructs it is necessary to run three different, though inter-related tools.

In terms of the number of different possible non-conforming constructs **mcc** detects the lion's share. The tool with the least number of different messages is **mcl**, and **mce** is likely to flag the same construct many times. The following is a rough guide as to what gets flagged by which component tool:

mcc	Operates on a single source file. Flags syntax and constraint errors. Also flags those undefined, implementation defined behaviours, unspecified behaviour and exceeding minimum limits that do not require knowledge of the values of objects (unless constant expressions are used).
mcl	Operates on one or more translation units that have been processed by **mcc**. Detects undefined behaviour resulting from inconsistent declarations and definitions across multiple translation units.
mce	Operates on the output file generated by **mcl**. Detects undefined and implementation defined behaviour occurring at runtime. This tool can be found in the dynamic portion of the **OSPC** tool set.

A strictly conforming program is one that is capable of being processed by **mcc, mcl** and **mce** without any warning messages being issued.

3.6 Becoming conformant in increments

Taking an existing program and making it strictly conforming is likely to require some effort.

The following series of actions will probably maximise the use of resources:

1 Use **mcc** to process all the source files making up a program. Providing no errors are reported files suitable for processing by **mcl** will be created.

2 The warnings generated as a result of (1) should be examined. Those that are considered as being unimportant, for the time being, can be ignored. The constructs causing the other warnings should be corrected.

3 Process (cross unit check) all the files, created by **mcc**, making up the program.

4 Any warnings generated as a result of (3) should be corrected. The path (1), (2) and (3) should then be repeated and so on until **mcl** does not generate any warnings.

5 Execute the program using **mce**. Correct any constructs that are flagged.

6 Make sure that any construct modified as a result of the previous phase do not cause **mcc** or **mcl** to flag additional warnings.

During the early stages of processing it might also be worthwhile to switch off some of the less important warnings. This can help reduce the volume of output and create a more manageable task.

3.7 Conforming to an API

API's (Application Program Interface) have become the method by which vendors define the software interface to their products. The product could be a piece of hardware, a third party library or even an operating system.

Users of applications often need to know which API's an application relies on (for instance when purchasing hardware and software separately). Managers of development teams would probably like to know that only the defined API is being used and that the interface rules laid down in the specification are being followed (to reduce the likelihood of their product becoming tied to a particular version, or vendors implementation of an API).

Running **OSPC** with the `-API` option switched on, will produce a listing of the API's used by the application and a summary of API specification violations.

3.7.1 What might an API define?

Information can be passed through an interface via function calls or external variables. To hide implementation details symbolic names (macros) are often used to represent special numeric values and structures are used to hold a collection of variables in one object.

The names of these functions, objects, macros and types are defined in one or more header files, to be included within the developers source code.

To be of use the API must define more than the C syntax. It must define the properties of these names. For instance the external `xyz` represents a status flag and can have any of the values given by the macros `A`, `B` or `C`.

3.7.2 Which API's are used?

Two things need to be done to answer this question:

1 Scan the applications source looking for all uses of external identifiers.

2 A database of API's and the identifiers they define against which identifiers used in an application can be matched needs to be available.

All references to identifiers are matched against the contents of the API database, or other parts of the application (one unit may refer to an identifier defined in another unit, not an API). A match against an identifier contained in an API flags that API as being used (cases where different API's define the same identifier are rare and can usually be resolved by looking at the context, ie included headers and the use made of the identifier).

Identifiers that are not contained in another unit of the application or the API database are regarded as referring to an unknown API (they could equally be referring to vendor extensions or a particular API).

3.7.2.1 Optional components

An API is sometimes broken down into core and optional components. For instance the ODBC has a core and two optional levels; the real time portion of POSIX has 16 optional components. The availability of these components can be tested for using feature test macros within the application source code.

To be useful, any report of API usage has to list those optional components of an API that are used by the application.

3.7.3 Are the interface conventions obeyed?

It is no good making use of the facilities provided by an API if the interface conventions are not followed. The whole purpose of an API is to isolate implementation details from the application. An applications that does not follow the specified interface rules is likely to have problems when using a new version of a library implementing the API, or the application is moved to a different platform.

So as well as finding out which identifiers are used, it is also necessary to check that they are used correctly.

3.7.4 Interface requirements specified in APIs

API's specify a number of different requirements for correct usage. Commonly seen requirements include:

1 Parameters

  a) Symbolic values must be used as arguments

  b) Types of arguments must be compatible with a defined type

2 Function return values

  a) Value has a properties, ie is positive, is negative

  b) Value may only be compared against symbolic values

  c) Feature test macros

    d) Used to check availability of optional constructs

3 Variable types

    a) Fields available in structs

    b) No requirement on layout of fields

    c) No requirement that the type be scalar

4 Not always constant

    a) Macros need not evaluate to a compile time constant

5 Headers

    a) Must be included

3.7.5 Function calls

An API function may accept input arguments, return a result or set status flags (for instance `errno`).

Functions that may perform various operations usually take an argument specifying which operation to be perform. For instance in:

```
fseek(file_ptr, 4, SEEK_CUR);
```

the third argument tells `fseek` that the seek is to be performed relative to the current file position. `SEEK_CUR` is a macro whose value will chosen by each implementation. The call:

```
fseek(file_ptr, 4, 1);
```

relies on an implementation choosing a `SEEK_CUR` value of 1. As such it does not obey the API specification, even though it will work on one or more implementations.

Some API's have more complicated requirements. For instance the POSIX function `open` may take one of three values (`O_RDONLY, O_WRONY, O_RDWR`) combined with zero or more other values (`O_APPEND, O_NONBLOCK, O_NOCTTY, O_TRUNC, O_CREAT, O_EXCL`). An API checker must ensure that the argument is created using the correct boolean or of these macros.

Checking a variable that is passed as an argument is substantially more difficult. It requires full flow analysis to track the symbols assigned to that variable. The current release of **OSPC** does not perform such analysis in this context.

APIs also define the types of the function parameters. Provided the host compiler supports function prototypes then the arguments given in calls to API functions will be checked at compile time. It is ok to pass an argument of a different arithmetic type because the compiler will automatically insert a cast to the required type. For instance if `size_t` has a `unsigned long` type, passing an `int` argument will work because of the implicit cast inserted by the compiler. Thus the developer does not have to worry about inserting casts to `size_t` for all appropriate arguments.

Passing an incorrect non-arithmetic type will cause the compiler to generate a compile time error. It is useful for an API checking tool to check that the arguments are compatible with the declared parameters, but not essential.

Some company coding standards require that arguments passed to functions are ‘strongly compatible’ with the argument type. That is the named types must matched. But this is a coding standards requirement, not an API requirement (because of the implicit casts inserted by the compiler).

API functions may also return values. These values may represent individual values or particular properties, such as positiveness. For instance `printf` returns the number of characters printed or a negative value if an error occurred.

```
if (printf(“abc”) == 3) /* OK */
   ;

if (printf(“xyz”) == -1)
   ;
```

The first example is checking the number of characters written against the expected value, as allowed by the API. The second is assuming a particular value for the property of negativeness. One implementation may return -1, another -2, another an arbitrary negative value. The correct test would be:

```
if (printf(“xyz”) < 0)
   ;
```

here the relational operator is testing for the negative property.

A grey area of checking involves functions that return a limited range of values. For instance the `tm_sec` field of a `struct tm` may take on values between 0 and 61. Is the following code fragment relying on an implementation defined extension or is it a coding bug?

```
if (t.tm_sec > 61)
   ;
```

OSPC assumes that it is a coding problem and does not flag this code as not conforming to the API.

Like arguments, return values may sometimes be symbolic.

```
if (fflush(file_ptr) == EOF) /* OK */
   ;

if (fflush(file_ptr) == 1)
   ;
```

The second example is incorrect because it assumes a value for the symbol `EOF`.

Also relational operators may not be used in those cases where all the values returned by an API function are symbolic.

3.7.5.1 Status flags

Status flags set by API calls are usually there to provide additional information. For instance `errno` might be set to some symbolic value to indicate the type of a particular failure. Few API's require status flags to be checked by the application.

OSPC has the ability to detect that applications are checking status flags after an API call. However, it makes the assumption that such checking must occur in the first conditional statement after the API function call. This check is regarded as a coding standards issue, not an API specification requirement.

3.7.5.2 Use of objects defined in API's

Like function return values, objects often have limits placed on the values they may contained. `errno` is an example of such an object. It may be reset to zero by the application, or it may be tested (using an equality operator) against a variety of symbolic names.

3.7.6 Optional constructs

Use of an optional API construct must be protected by a feature test macro. For instance POSIX specifies that the function setuid is only available if the ftm `_POSIX_SAVED_IDS` is defined. The developer thus has to write the code:

```
#ifdef _POSIX_JOB_CONTROL
 setuid(23);
#else
 do_something_else(23);
#endif
```

here the code is checking for the availability of `setuid` and taking alternative action if it is not available.

Optional constructs may be any identifier declared or defined by the API.

Developers that are unaware they are using optional constructs have set a future trap in the porting of their application. Users of packages also need to be aware of any optional constructs required by an applications when specifying hardware or third party libraries.

3.7.7 Use of headers

Headers are the means by which identifiers defined by in API may be made visible to the application. In some cases the header must be included because it contains information that cannot be obtained elsewhere (for instance the values chosen by the implementation for symbolic names). Sometimes it is possible for a developer to declare a subset of the API without including the header.

Headers are necessary if symbolic macros and types are referenced from the application source. For instance in the example involving `fseek` above the header `stdio.h` needs to be included so that the compiler can obtained the value of the macro `SEEK_SET` chosen by the implementation.

An example where an API header need not be include is the `strerror` function. It is ok to declare that function explicitly, rather than including the `string.h` header. Because its API specification only uses C predefined types, `char *strerror(int errnum)`. However, `memset` could not be so declared in the users source without including the `string.h` header (if the header is included why explicitly declare it anyway). This is because the declaration of `memset` needs a type from that header, `size_t`. The developer may declare `memset` with a particular predefined type instead of `size_t`, but that will only work on implementations where that type is used to represent `size_t`. (The C API specifies the type `void *memset(void *s, int c, size_t n)`).

3.7.7.1 Incorrect header contents

A problem that sometimes arises with API headers is that they do not accurately reflect the requirements contained in an API. Fortunately the most common problem, incorrectly specified parameter arithmetic types, does not affect the performance of a checking tool. If, for instance, a vendors version of `string.h` declared the third parameter of `memset` to take an `unsigned int` argument the interface is not broken from the applications point of view, provided `size_t` is also declared to have type `unsigned int`. The compiler vendor is at fault for not upgrading its headers to conform to the C standard (first published in 1989 by ANSI and as an ISO standard in 1992).

Other problems often seen include syntax violations (text after a `#endif` not included within comment delimiters for example) and incorrect numeric value for macros (floating point values inaccurate in the last digit).

3.7.8 Use of API defined types

An API may define types to allow implementations to adapt themselves to different hardware (usually different sized scalar types) or to combine together similar variables in one place (a struct).

APIs rarely define the ordering of fields within a struct, although implementations are usually given liberty to add additional fields to structs. Applications that rely on ordering of fields or make use of implementation specific fields are going beyond the specification given in the API.

3.7.8.1 struct fields

Initialisation of struct objects, via an initialiser, is one example where an ordering of fields is implied. So the construct:

```
div_t local_var = {1, 2};
```

must be explicitly expanded out to (assuming the above assumed this order):

```
div_t local_var;

local_var.quot=1;
local_var.rem=2;
```

An example of an implementation adding additional fields to a structure is `struct dirent`. On a Sun platform the code fragment:

```
if (dirent_obj.d_reclen == 3)
   ;
```

would happily compile. Other platforms are likely to complain that the field `d_reclen` does not exist (it is not in the POSIX or XPG API's).

3.7.8.2 Type need not be scalar

An API occasionally leaves the specification of a particular type wide open. An example is the `fpos_t` typedef specified in the C standard, which simply states "... which is an object type capable of recording all the information needed to specify uniquely every position within a file." On many systems this type is a scalar. So the code:

```
if (fp_1 == fp2) /* two variables of type fpos_t */
   ;
```

works. But C does not allow the `==` operator to be applied to struct types. This code fragment would fail to compile on a platform that defined `fpos_t` to be a struct (in fact there is no portable way of comparing two objects of arbitrary type for equality).

3.7.9 Symbolic name need not be constant

API's use symbolic macro names to represent values that may vary between implementations. Developers sometimes assume that because macros are used the value will be a constant literal. This is sometimes not the case. For instance, of all the macros used to describe properties of the floating point representation, in the C standard, only one, `FLT_RADIX`, is required to be a constant expression. On many implementation the other macros are indeed constant expressions, but they are not required to be.

The code fragment:

```
#include <float.h>

int number[FLT_DIG];
```

relies on `FLT_DIG`, the number of decimal digits in a number that can be exactly represented in a float, being a constant expression. If it is an expression that must be evaluated, as above, at runtime the compiler will not be able to compile the application.

3.7.10 Implementation specific and future problems

API's often reserve specific names for future releases of their specification. They also allow implementations to add additional names to headers, provided those names obey a few restrictions.

The presence of these reserved names effectively constrains an application from defining identifiers with those names. An application containing such a definition could fail to compile on certain platforms, or with later versions of the API (because of duplicate or inconsistent definitions).

Some reserved names are easy to avoid by the application developer, for instance those starting with double underscore. Others might be considered more contentious. For instance all macros starting with `E` (capital E) are reserved by the C standard if the header `errno.h` is included, and all external identifiers starting with the three characters `str` are reserved in all cases by the C standard.

Experience has shown that applications often contain definitions of many identifiers whose names clash with those reserved by API's. The definitions could be changed to use alternative names, but in many cases the effort involved is disproportional to the time and effort needed to modify existing code.

Insisting that all names defined by an application not clash with those reserved by the API's used is impractical. Instead an API checking tool should list all definitions that do clash, along with a count of the number of references to them. Applications vendors might also undertake not to add new definitions to this existing list.

3.7.10.1 Identifiers specified to have properties

The C standard defines `errno` to "... expands to a modifiable lvalue that has type int ... It is unspecified whether errno is a macro or an identifier declared with external linkage." This is an example of an API defining properties of an interface rather than C syntax for its implementation. The POSIX standard says "... which is defined as extern int errno;" An implementation specification.

This kind of API object specification, using properties rather than C syntax is not very common.

Ideally an API checking tool would know about the different properties defined by an API and flag discrepancies. From the tools point of view such special cases are just that, special cases. In the example above it was reasoned that few applications rely solely on the C standard, most also include POSIX (or an API based on POSIX). So checks suggested by the specification given in the C standard is not carried out by **OSPC**.

3.7.11 Use of identifiers of unknown status

Once the entire application has been processed all referenced identifiers are known. Resolving these identifiers against those defined by the application and those defined by the known API's may leave some unaccounted for.

These unaccounted identifiers are assumed to belong to either an unknown API or extensions to a known API.

In the case of variables and macros there will be a declaration in one of the included headers. The name of the header may give clues to the status of the identifier.

Functions need not be declared prior to use. In this case the compiler will create a default declaration of `extern int f()`, where `f` is replaces by the name of the function. So there may not be a header name to refer to for guidance. Once again incorrectly written headers can confuse the analysis. It is not unknown for vendors to supply headers with some function declarations missing, even though code implementing that function is available in a library that can be linked against.

A checking tool can do no more than list identifiers whose status is unknown. This list may contain hints as to their likely status, for instance by giving the name of the header in which any declaration occurred.

3.7.12 Identifier specified by several API's

Sometimes a newer API will add functionality to an interface defined by an earlier API, or define what was previously undefined behaviour. For instance the C standard says that the `rename` function may be used to change the name of a file. But C has no concept of directory structure, so it does not include any specification for handling directories, it assumes a flat file system. POSIX defines a directory structure and adds to the specification to rename to describe how directories are to be handled.

It is not always possible to deduce the use being made of an API interface from static analysis of the source. **OSPC** takes the view that if an identifier from an API is referenced then that API is used, irrespective of the number of API's involved.

3.7.13 Can all referenced API's be detected?

No, they cannot. Consider the case of an API that only defines macros and types. Let us assume that information on this API is not available to **OSPC**. Who is to say that the included header, used to access the defined names, is not part of the application, rather than an API? An example of of a header that only contains macros and typedefs is `stddef.h`, from the C standard.

In the case of objects and functions their definition is contained in a library, not in the source making up an application. Such a library has the opportunity to modify an object and functions may access host specific information.

Does it matter that use of an API may go undetected? Perhaps not. The developer has the option of taking the headers containing the macro and type definitions and making them part of the application source tree (if they are not available on a given platform). Of course the developer then has to take over responsibility for ensuring that the definitions are correct for each new platform.

In the ideal case the **OSPC** database contains information on an API's that an application uses.

3.7.14 Information output by an API checking tool

End users need a list of the API's used, along with a summary of any discrepancies. Software developers would probably like any discrepancies to be pin pointed, simplifying the job of isolating and fixing them.

Being able to output a list of applicable API's relies on having a database of information about what each API contains. This is turn requires an API to be documented, which, unfortunately is not always the case (X11 being an example, where even the headers provided can vary between platforms, let alone the header contents).

3.7.14.1 Information summary

1 API's used

a) Optional components used

2 Violations of the defined interface

a) Type of violation and number of occurrences

3 Reserved ids used

a) API that reserves them

b) Identifier name and number of references to it

4 Identifiers referenced that are not in a known API

a) included header

b) functions

c) external identifiers

d) macros

Chapter 4

User interface and configuration

4.1 Introduction

All the components of the **OSPC** share the same user interface. This chapter gives an overview of this user interface. There are four basic components:

1. Command line options
2. Configuration files
3. Profiles
4. Tool configuration files

Some options are common to all component tools, while others are specific to a given tool. A list of options available for a given tool, together with their default values, can be obtained by typing the name of the tool on its own at the command line.

4.2 Command line options

Arguments on the command line may be mixed in any order and have few restrictions on their format. All command line arguments are processed before the component tool starts to analyse the user's file. Thus the following three commands all have the same effect.

mcl file1 -LOG- file2 -LOG+ -REF+ file3

mcl -LOG+ -REF+ file1 file2 file

mcl -VIA f.via file1 file2 file3

where `f.via` contains the lines:

-REF+

-LOG+

Thus it is not possible to switch options on and off for specific files. Lines in via files are processed as if they had occurred on the command line, not separated by new-lines (details of the `-VIA` option are provided later).

Any sequence of characters immediately proceeded by a "-" (minus) character is treated as a command line option. Each option processes the tokens that follow it in a predefined manner. All other character sequences are treated as names of files.

A command line option can take one of the following forms:

1 Option without any other information.

  -HELP

2 Option with a numeric parameter.

  -NAMelength 31

3 Option with a string parameter.

  -ECHO "Hello world"

4 On/Off option.

  -LOG+

Zero or more whitespace characters may separate options from their numeric or string parameter. Numeric and string options may consist of a list of values (e.g., `-I`) or a single value (e.g., `-NAMelength`).

4.2.1 Abbreviating options

The user need not type the complete option name in some circumstances. Options may be abbreviated. This abbreviation takes the form of missing off characters from the end of the option name. Any number of characters may be omitted, provided sufficient remain to disambiguate the option intended from any other possible options. The minimum abbreviation is given by the portion of the command in upper case on the help display.

mcc util -ARith+

mcc util -AR+

mcc util -ARithrsh+

All the above have the same effect.

4.2.2 Numeric parameters

The format of numeric parameters follows the same rules as C literal constants. They may be given in decimal, hexadecimal, or octal.

mcl -NAMelength 0x3000

mcl -NAMelength 030000

mcl -NAMelength 12288

All have the same effect. The maximum value possible for a particular option can be obtained by specifying the keyword `MAX` rather than a numeric value.

mcc prog -NAMelength max

4.2.3 String parameters

These take the form:

1. A normal string. Here the string is terminated when the first whitespace character is encountered. These are represented by the notation `<letter-sequence>`.

2. The name of a file. Like (1) it is terminated when the first whitespace character is encountered. The current path prefix (if any) as added to the front of the filename. These are represented by the notation `<filename>`.

   mcc prog -O /usr/demo/Example1/util.kic

3. Argument to `-SHStart` or `-SHEnd`. The arguments to these options are delimited by a single character. The character used is the first non-whitespace character seen after the option name. The string is terminated when the same character is next encountered. These are represented by the notation `<string>`.

   mcc prog -SHStart !ps ax!

4. Argument to `-ECHO` or `-REM`. These options cannot occur on the command line. When they occur in an options or via file the argument consists of the rest of the line. These are represented by the notation `<string>`.

   -REM Profile for a 68000 cpu

   -REM

4.2.4 On/Off options

Options taking this form are followed by:

1. '+' turn option on

   mcc -Lis+

2. '-' turn option off

   mcl -Lib-

3 if no character follows the option, a ‘+‘ is assumed and the option is turned on

mcc -Q

Note: No whitespace is allowed between the option name and the **+**/**-**.

4.2.5 Default option settings

If a configuration or local options file is available and has been successfully read then the default values will be picked up from that file. Otherwise each component tool contains internal default values which are set prior to reading the configuration file (the default default settings).

The settings of the default values currently in force are displayed next to, or below, the option name when the help screen is given.

For details of the structure of the default options, configuration strings and error files, and a discussion on how to change them see Chapter 2 in the User Reference Guide.

4.3 Local options file

Sometimes a user may want a different set of options to be in force while processing a particular group of files. One approach would be to make a local copy of the options file and to modify the options setting in that configuration file. The `-CONFIG` option could then be used to cause this new option file to be read. Rather than having to give the same command line arguments for every input file it is possible to create a local options file.

A local options file may contain a series of command line options. This local options file is read after the default options have been read. Thus it overrides any settings made in the options file.

In turn, any command line options take precedence over the settings given in the local options file.

The local options file is search for in the current directory. Its filename is generated by appending the letters ‘rc’ to the component tool name and prefixing a ‘.’ character, i.e., `.mccrc`.

4.3.1 Creating a local options file

The local options file is a text file that can be created using a text editor. It should contain one command option per line. These options may add to, or override options contained in the default options file.

-I /usr/me/myheads

-List+

In this example the `-I` option causes the path `/usr/me/myheads` to be added to the list of places to be searched when looking for header files. The default for the `-List` option may be either on or off. Here we are overriding the default setting to switch the listing on.

The `-Forgetall` option may be used to undo the effects of previous options.

-Forgetall nomsg

This line causes all previous message suppressions (the effect of the `-Nomsg` option) to be reactivated, i.e., the behaviour is the same as if they had never been given.

4.4 Error reporting

Each component tool has uses the same error reporting machinery. Because of its capacity to report on such a wide range of problems the output from **OSPC** can sometimes be overpowering. A range of options has been provided to enable the user to control the generation of warning messages. These include the ability to:

- Stop checking after a given number of errors/warnings
- Suppress specific errors/warnings
- Suppress warnings below a given level
- Switch to strict C standard mode

Each component tool also has error reporting options that are specific to its role in the checking process. These are described in the chapter on that tool. The following options can be used with any of the tools.

Sometimes a run of **OSPC** results in a large number of error messages occurring. This can happen because there are large numbers of problems in the source or because of some syntax error that **OSPC** failed to recover fully from. The `-MAXErrors` option enables the maximum number of allowable errors to be specified. Once this limit is exceeded **OSPC** stops processing the current source file. The `-MAXWarnings` option is similar to `-MAX-Errors` except that it applies to warnings.

<tool> prog -MAXE 55 -MAXW 200

Rather than halting **OSPC** after a given number of warnings, analysis of the messages may show that the majority are caused by a few constructs. The `-Nomsg` option can be used to switch off individual error or warning messages. Once a given error or warning has been switched off it will not appear during the current invocation of **OSPC** and will not contribute to the message count.

Note: Disabling fatal errors serves no useful purpose. Any messages given after a suppressed constraint error may be difficult to interpret without seeing the constraint message.

To disable warning numbers 43 and 97 type:

<tool> prog -N43 -N97

Another method of reducing the number of warnings produced is to cut off those below a certain level. The error message file associates one or more numeric levels with each error number it contains. The number given in the `-SUppresslevel` option can be used to act as a cutoff. Messages with levels below the value given will not appear in the output.

<tool> prog -SU6

Thus if the highest level specified for a given message level is 5 and the cutoff is 6 no messages will ever appear for that error number. If messages at levels 5,6 and 7 are available for a given construct and the cutoff is level 6 then the level 6 message acts as the minimum level available.

By default **OSPC** is tolerant. It assumes the lowest severity message possible. It is possible to reverse this behaviour. The `-STandard` option does not cause **OSPC** to perform any more checks. Rather it causes the messaging system to use the highest level of message available. The default behaviour is to attempt to recover from errors and use the lowest level message possible. When the `-STandard` option is switched on recovery still occurs, but the highest level message possible is given. It also disables the use of extensions.

<tool> prog -ST+

If messages at levels 5,6 and 7 are available for a given construct. Switching on the `-STandard` option causes the highest available message, 7 in this case, to be given. If this option had not been switched on the message at level 5 would have been given.

Note: The error reporting machinery will never give the same warning on the same token. To be more exact, when errors are reported the system remembers what error numbers have already been given for each token. Once an error number has been reported for a given token it is not given again for that token. It is possible for the same error number to appear on adjacent tokens.

The messages reported by **OSPC** are intended to be informative and easily understood. Sometimes this aim may not have been achieved or the user wants further information. Switching on the `-REFerence` option causes the message reporting machinery to also give a reference to the standard (provided one is present in the error file) with the text of any messages generated.

<tool> prog -REF+

4.4.1 Locating the error files

Given that each tool and standards profile has its own associated error file it can sometimes be difficult to find out where particular error messages are kept. The **errorrange** command (actually it’s a script) will list each error number and the file where it can be found.

4.5 Platform profiles

Platform profiles are a method of encapsulating all the relevant information about a given computing platform. As well as providing a convenient tag for describing a given platform they can also be used to give more specific warnings about non-portable constructs. By knowing the source and target platform it is possible to filter out those warnings that are not applicable. Those warnings that are generated are specific to porting the software from one machine to another.

The information relating to each platform is held within a group of directories. The organisation of these directories is derived from the underlying components that create a platform. These include the cpu, compiler and O/S. By organising the information in this fashion it is possible to reuse profiles, once created. For instance once the Sparc cpu profile has been created it can be referenced from any profile that contains a Sparc cpu.

Each component tool of **OSPC** has a large number of options. Most of these relate to technical details of cpus and compilers. Platform profiles allow these details to be hidden. The default help text excludes those options, for a given tool, that are dealt with through platform profiles. These options can still be given on the command line and a full list of the options can be obtained by using the `-DETail` help option.

It is possible to create, modify and delete platform profiles. This is best done using the **profadm** script. Full details on this tool can be found in the User Reference Manual. Type Bprofadm to get obtain a list of possible options.

4.5.1 Paths

In several situations the path used to locate a file is deduced from information provided by the system.

The `checkinfo` directory is located by finding the path along which the tool currently being executed was found. Thus if that tool is moved to another directory it is also necessary to move the `checkinfo` directory.

The default include path and the location of the file `lib.klc` are kept in the file `checkinfo/host/locate`. This file is created by the **doinstall** script.

The only other mechanism used to locate files is through the Unix environment variable `PATH`. This follows the standard Unix rules.

4.6 Common options

The following options are common to all tools. In some cases these options have been extended within a given tool. Where this has happened details on the extension are provided in the chapter dealing with that tool.

4.6.1 Changing the configuration

The configuration of the **OSPC** is 'soft'. That is the options are read in at startup, rather than being compiled in. This flexibility allows it to support a wide range of architectures and compilers.

At some time users will want to change the default configurations. The `-COnfig` option provides a means of doing this. Since there are multiple configuration files the parameter to this option specifies which configuration is being changed. The common configuration files are:

- 'strings'. The strings for all the output produced by the tool.
- 'options'. The default option settings.
- 'locate'. The location of the various special files, e.g., libraries.

Example

mcc prog -config strings=/home/usr/fred/misc/newstrings

Each tool may also have its own individual set of configuration files. These are described more fully in the chapter on that tool.

If the file cannot be opened, or is not in the correct format an error message is displayed and processing stops.

The layering of the configuration option processing can sometimes result in an option obtaining a value that needs to be got rid of. For options taking single values it is simply a matter of specifying a new one. However, some options build up a list of values. For instance an `-I` option does not undo the effects of any prior `-I` option. It adds a new path to the list of existing paths.

The `-Forgetall` option provides a means of 'loosing' these values. It must be applied to an option that takes lists of values, or a single string. The following options are some of those that may be 'forgotten':

LOGfile	Cancel request for logfile.
NOmsg	Reinstate any previously suppressed error numbers.
Output	Use default output filename.
PAth	Cancel previous prefix.

Each tool may also have its own individual list of option values that can be forgotten. These are described in the chapter for that component tool.

It is sometimes the case that certain command line options are frequently used with a given set of files. Such a sequence of options is unlikely to warrant being placed in the local configuration file. Another possibility is to place the options in a file and cause that file to be read from the command line. The `-Via` option provides just such a facility.

Via files are text files, created by the user, that contain frequently used command line options and their parameters. Options in a via file are given one per line. A via file may contain a reference to other via files (to an arbitrary depth, subject to the maximum number of files that can be open at any instant). Each reference will be processed as it is encountered, and when the end of the file is reached processing will continue in the original file with the line after the `-Via` option.

All output generated by **mcc** is sent to `stderr`. The `-LOGfile` option can be used to specify the name of a file to which the output should also be written.

```
mcc prog -LOGfile results.log
```

Causes all output that is sent to standard output and standard error to be sent to the file `results.log`.

4.7 Environment variables

OSPC follows the same conventions as the Unix shells in the handling of environment variables. Identifiers that begin with a `$` character are substituted by the value of the environment variable of the same name. Thus if the options file contained the line:

```
-i$INCLUDE
```

and the environment variable **INCLUDE** had the value `/usr/include` the effect would be the same as if:

```
-i/usr/include
```

had been written.

The handling of the `$` character in via and options files is intended to mimic the shell expansion on the command line. So that a use of `$` as an argument on the command line will have the same effect if placed in a via or options file.

A useful point to remember, for creating unique file names, is that `$$` expands to the process number of the tool currently being executed. Also shell conventions for environment variable expansions can be used, as in:

```
-i${ROOT}/include
```

4.7.1 Predefined variables

The system environment variable `HOME` is used to locate the .rc files.

Each tool may make use of other predefined environment variables. These are described in the chapter on the respective tool.

4.8 Order of reading setup information

Each tool obtains information on options from several sources. In order to know how these interact it is necessary to know the order in which the various options are processed.

The profiles, local config file and command line options are processed in the following order:

1. Host, source and target are assigned default-default values
2. Command line is read, and immediate options are processed
3. Host/config default file (`mcc/options`) is read
4. The file `.mccrc`, located along the `HOME` path file is read, and overrides host information.
5. Any `.mccrc` file in the local directory is read, and overrides host information.
6. Source default file (`mcc/srcopts`) is read
7. Target default file (`mcc/tgtopts`) is read
8. Source platform profile read
9. Target platform profile read
10. Target options overrides host information
11. Command line options override host information.

4.8.1 Processing the component profiles

When a platform profile is read, its components are read in the order :

1. cpu
2. abi
3. standards
4. os
5. compiler

6 misc

This reflects the natural order of precedence: Compilers have ultimate control over the code that is produced, and how it interfaces to the O/S etc. An ABI can be more specific than a cpu definition. An O/S, and compiler, may choose whether or not to implement completely, the standards they proport to support.

Options that are not given explicit values retain their default default values.

4.9 Integrating with other tools

How the **OSPC** tools process their options has obviously been influence by existing Unix conventions. In particular the **cc** and **ld** commands have influence the choice of options and the action that they perform.

All tools provide return codes. The conditions required to return a given value follow the same conventions as other Unix tools. Use of return codes is necessary for the make file machinery to work correctly. The range of possible return codes and the conditions under which they are given are described in the chapter on the respective tool.

Chapter 5

OSPC source checking

5.1 Introduction

The name of the **OSPC** tool that performs checks on the source code is **mcc** (Model Implementation C Checker). **mcc** takes a C source file, known as a translation unit, performs syntactic and semantic checks and generates a .kic file. This .kic file can be processed, along with other .kic files, by the cross translation unit checker **mcl** (Model Implementation C linker) to detect cross unit inconsistencies. The C source file may include other source files. Other input to the compiler includes information on the source and target platforms. The default configuration is for **mcc** to operate as much like the host C compiler as possible. Thus both existing make files and scripts can easily be used.

During startup **mcc** attempts to locate various configuration files and read the options they contain. If these configuration files cannot be located internal default values are used. While processing the source, the main difference from using the development compiler, typically **cc**, that will be noticed is that **mcc** gives significantly more warning messages. This difference is to be expected since **mcc**'s basic job is to give the user information about potential problems in the source.

For obvious compatibility reasons the functionality provided by **mcc** and its command line options have been influenced by the Unix **cc** command.

5.2 Using mcc

This section gives a quick overview of the more commonly used **mcc** options. A full description of all options can be found in the User Reference Manual.

Mcc is invoked by typing **mcc** followed by the name of the file to be compiled:

```
mcc file
```

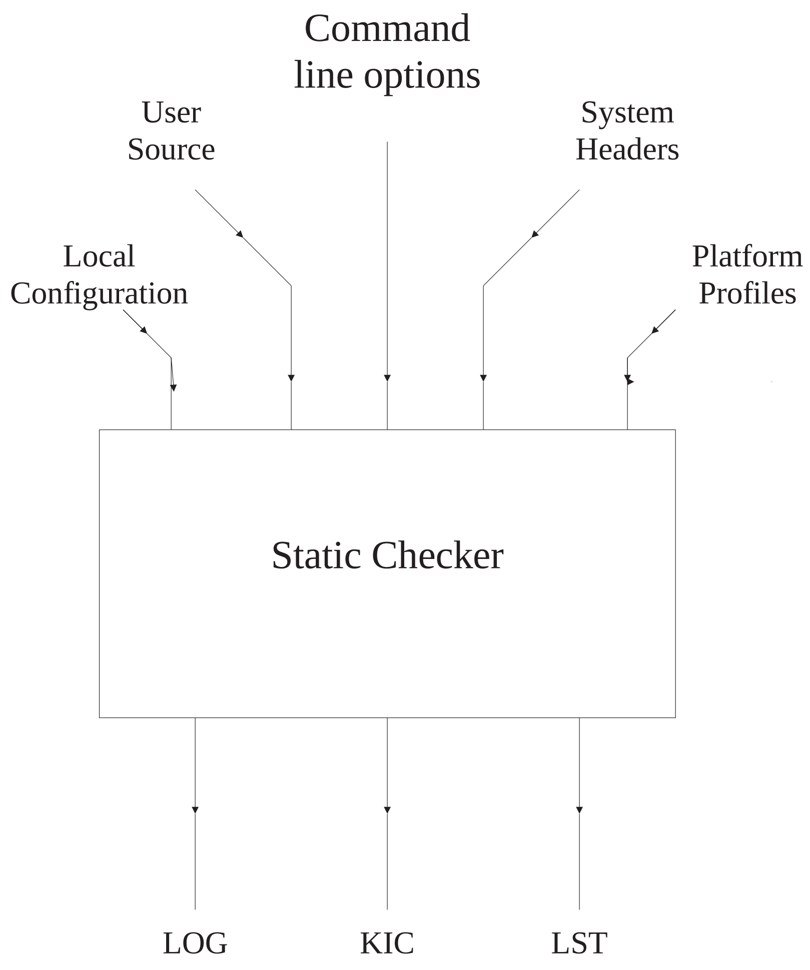
mcc will add the suffix .c if one is not given.

Mcc can process more than one file at a time. It also accepts a range of options. To obtain a list of these options (help text) type:

```
mcc
```

or

```
mcc -help
```



Static checker input and output files

Probably the two most important options given to **mcc** are the source and target platform profile names.

mcc prog -src sun4

The `-SOurce` option specifies a platform on which the program is known to compile and execute correctly. If no target platform is given the default is used. The default target platform can be overridden by specifying one on the command line.

mcc prog -src sun4 -tgt cabstract

Where `cabstract` is the name given to a platform profile that follows the exact letter of the ISO C standard (a full list of supported platforms can be obtained by using the **profadm** command with the `LIST` option). If the name given for the platform profile cannot be found a warning is given and the internal default values used.

The platform profile information is used by **mcc** to reduce the number of 'uninteresting' warnings generated.

The warnings generated by **mcc** only give local context. If a source line does not contain a warning it is not output. The `-Listing` option can be used to cause messages to appear in their context. Switching this option on causes a list file to be generated. A listing file contains the preprocessed source of the input file plus any associated error or warning messages. The source file name is used to create a listing file, by substituting a .lst for the .c suffix (or implied .c if none is given).

mcc prog -l+

will cause a list file called `prog.lst` to be generated. This listing will contain the text of the preprocessed source along with any messages that might have been generated. If an exact copy of the

mcc prog -trace input

input source is required the `-TRACE input` option should be used. This will cause each line of input to be sent to standard output, as it is read in.

A related option is `-PPlist`. This option is similar to the `-Listing` option, but with the difference that the file produced is compilable (assuming that there were no constraint errors in the original). A .i is appended to the output filename.

mcc prog -pp+

creates a file called `prog.i`. All the preprocessor directives will have been expanded. That is all include files will have been included, all macros expanded and the conditional compilation options selected. The main use for this option is in helping to pin down those errors that are hard to spot in the original source, because of heavy use of preprocessor directives.

Command line options may also affect the way that the source code is processed. Two such options are `-D` (for define) and `-I` (for include).

The `-D` option allows the user to define C macros from the command line. These macros are treated as if they formed the first lines of the source file being compiled. There are two forms of this option:

-D ident is equivalent to placing the C preprocessor directive:

#define ident 1

at the beginning of the file being compiled. While:

-D ident=string is equivalent to the C:

#define ident string

Note:

-D ident= is equivalent to:

#define ident

Whitespace may not be used to separate the = character from the preceding identifier or following string.

The `-I` option causes the specified directory to be added to the list of directories used by the `#include` file search algorithm.

mcc prog -i../common

By default the name of the input source file (striped of any associated path) is used as the basis of the generated output .kic filename. The `-Output` option causes the given name to be used as the .kic filename.

mcc prog -o newfile.kic

causes `prog.c` to be compiled and the name `newfile.kic` to be used to hold the generated code and symbolic information.

If more than one source file is being processed the `-OUTPUTPath` option can be used to specify a directory other than the current to which the output should be written.

mcc prog1 prog2 -outputp ../temp

Some language extensions have become so common on certain hosts that the majority of compilers on those hosts support them. On DOS these extensions include the keywords *near, far, huge, pascal* and *fortran*. The `-EXtensions` option tells **mcc** to enable the handling of specific, known, extensions. How these are handled varies according to the extension itself. Full details can be found in the User Reference Manual.

mcc prog -ex dos

The other major compiler extension support via this option is SVR4. The `#assert` functionality is commonly used in the system files on this release of Unix.

An alternative technique, for DOS at least, is to use the `-D` option to ‘null out’ the presence of these tokens.

mcc prog -Dfar= -Dnear= -Dhuge=

5.2.1 Error reporting

With the ability to report over 1,500 different problems in the source code the output from **mcc** can sometimes be overpowering. A range of options has been provided to enable the user to control the generation of warning messages. As well as the common error control options there is one option specific to **mcc**.

Not all of the source code seen while processing a translation unit is under the user’s control. Sometimes it may be necessary to use a header file other than those supplied with the **OSPC**. These alternative headers, perhaps provided with the development compiler, may not be strictly conforming. Declarations and definitions within these headers may therefore be flagged by **mcc**. However, the user does not always want, or have the option of changing these headers.

Switching the option `-HDRsuppress` on causes **mcc** not to display any messages about the contents of systems headers.

Note: A constraint error within a system header will still be flagged.

mcc prog -hdr+

Sometimes a run of **mcc** results in a large number of error messages occurring. This can happen because there are large numbers of problems in the source or because of some syntax error that **mcc** failed to recover fully from. The `-MAXErrors` option enables the maximum number of allowable errors to be specified. Once this limit is exceeded **mcc** stops processing the current source file. The `-MAXWarnings` option is similar to `-MAXErrors` except that it applies to warnings.

mcc prog -maxe 55 -maxw 200

Note: Disabling constraint errors, using the `-Nomsg` option, serves no useful purpose. Any messages given after a suppressed constraint error may be difficult to interpret without seeing the constraint message.

The messages reported by **mcc** are intended to be informative and easily understood. Sometimes this aim may not have been achieved or the user wants further information. Switching on the `-REFerence` option causes the message reporting machinery to also give a reference to the standard (provided one is present in the error file) with the text of any messages generated.

```
mcc prog -ref+
```

While processing a source file **mcc** gives information on its progress. Switching on the `-Quiet` option causes the generation of this progress reporting output, including the reporting of errors to standard output to be disabled.

```
mcc prog -q+
```

5.2.2 Forgetall

In addition to the common parameters to this option **mcc** also supports:

Assert	Forget previous assertions
CODES	Forget previous coding standard enablings
D	Forget previous definitions
I	Forget include paths given so far
PAth	Forget previous paths
Size	Forget previous settings for sizes of objects
STRUCT	Forget previous STRUCT file names

For instance an `-I` option does not undo the effects of any prior `-I` option. It adds a new path to the list of existing paths. To change the order in which paths are searched it is first necessary to forget any previous paths. A new path ordering can then be specified.

5.3 Nonconforming constructs

So far we have discussed the basic **mcc** options that might be given on the command line. This and the following sections will deal with those aspects of **mcc** that fall into the realm of standards conformance checking. They deal with constructs that are perfectly legal ISO C code. But the use being made of these constructs puts them outside the bounds of another standard.

5.4 String contents

Source code that contains the full path names of files is unlikely to be portable. These path names are likely to be contained within strings. The identifier pattern matching functionality can also be used to search for patterns contained within strings (paths are simply the most likely use).

```
#include “/usr/include/ed.h”

char *ed = “/bin/vi”,

     *tell = “The editor is /bin/vi”,

     *what = “The file /bin/vital holds important information”;
```

In the above code several constructs could be flagged:

1 The use of absolute pathname in the i#include. directive.

2 The use of file names in strings.

3 The use of a particular string, for instance the word “file”.

See the section on `#strings` in the Reference Manual.

5.5 Status flags

As well as passing back information to the caller via a return result, or parameters; functions may also assign to global status flags. For instance many of the C library functions may set `errno`.

Many coding standards require that status flags set by function calls be checked.

```
x = sin(y);
if (errno == ERANGE)
   /* out of range error, do something */
```

While processing the C source **OSPC** notes any status flags that might be set by API calls (of course such information has to be available in the API database). If the status flag is not checked in an immediately following `if` statement a warning is given.

A warning is also given in the following case:

```
x = sin(y1) + cos(y2);
```

where each call may independently set `errno`. No warning is given if the calls could only have assigned one, identical value, to a status flag.

The API database can also hold information specifying that API calls check specified status flags. Calls to such functions are regarded as being equivalent to an `if` statement, i.e., the status flag is marked as having been checked.

See the section on `#status flags` in the Reference Manual for more information.

5.6 Sizes of datatypes

The C language is used to write application for a wide range of processor types. Depending on the power of the processor the sizes of the basic language data types may vary. The most commonly seen variations are in the *int* type. This might be 16, 32 and now 64 bits wide (the *char* type may also be larger than 8 bits on some signal processing chips, but this is not discussed here).

Objects of *int* type are probably the most common datatype used in a C application. It is also the default type for function return types and parameters. In an attempt to be ‘friendly’ C specifies many instances where implicit casting occurs. So developers are not required to insert explicit casts. In some cases an application may have assumptions about the size of this type built into it. **OSPC** cannot help in the area of algorithm redesign. But what it can do is flag those instances where implicit type conversions are likely to cause problems.

Probably the most common cause of problems are assumptions about the relationship between the sizes of data types. In particular between *int* and *long* (and sometimes between *int*, *long* and pointers). In the Unix world it is commonly assumed that *int* and *long* have the same size (and that *long* and pointers have the same size).

OSPC uses the cpu component of the platform profile to work out if there are likely to be problems in porting to the target processor.

In the following example:

```
func();
{
long l;
/* ... */

return l;
}
```

the function, *func*, defaults to returning an *int* type. However, the return statement contains an expression of type long. The C standard specifies that an implicit cast needs to be inserted and an *int* returned. If this code was written on a platform where *int* and *long* had the same size there is the possibility that the developer had been lazy and omitted the return type. **OSPC** will flag this function (given a 32 bit source and 64 bit target), suggesting that the return type might be *long*. Inserting an explicit cast on the return expression would make the intention clearer and no warning would occur.

Discrepancies caused by *int* and *long* types being mismatches across interfaces are flagged by the cross unit checker. This topic is discussed in the next chapter.

5.6.1 Using prototypes

When porting to a machine where the sizes of the scalar types are different from the source processor, prototypes are essential. A header that contains the declaration:

```
long a_func();
```

or even worse:

```
another_func();
```

says nothing about the parameter types. In the second case we can only hope that an *int* return type was intended.

If a function prototype is used the development compiler and **mcc** can use the information provided to ensure that calls to functions are as intended. Without prototypes a development compiler will simply follow the rules for default promotions and **mcc** will flag any incompatible calls. If prototypes are not available the developer will have to insert casts on the parameters of all calls whose type is not *int*.

5.6.2 Types of constant expressions

Unless given a prefix the type of an integral constant expression is governed by the nearest type into which it will fit (ok there are are some complications for hex and octal literals). When porting to a platform that uses more bits to represent integers than the source platform there are likely to be some changes in types. It is likely that literals that once had *unsigned int* type will now be represented as *int*.

Extra bits used in the representation could also mean that constant arithmetic that previously overflowed, or wrapped, will now yield the correct, larger, value. Such constant folding is common in conditional inclusion directives in the preprocessor.

Internally **OSPC** is capable of handling integral arithmetic to any precision (well actually 64k bits in a long). As shipped it is built to be able to handle up to 64 bits in an integral type. Thus those integral literals that will have a different type on the target platform, from the source platform, will be flagged. Also, when constant expressions are folded those cases where the result will vary, depending on the number of bits in the datatype, will be flagged.

5.7 Embedded SQL

Applications that make use of databases may interface to them via API calls or embedded SQL. In the case of embedded SQL database vendors provide preprocessors to convert the SQL into C prior to passing it through a compiler. A lot of information is lost in this preprocessing phase. Not to worry **OSPC** supports the embedding of SQL in C, using the `-SQL` option.

While processing the SQL basic checks are made to ensure the consistency of the statements, i.e., host variables given in an `INTO` clause are consistent in with the select expression. Also use and assignments of host variables is tied into uninitialised variable checking.

```
int some_global = 4;

void f0()
{
char val1,
     val2;
int i,
    k1,
    k2,
    pno1;

struct {
       int salary;
       int holiday;
       } emp;

EXEC SQL
```

```
   GET descriptor 'ext_cur_name' :i = COUNT ;

/* use of undeclared host variables flagged */
EXEC SQL
   GET descriptor 1 :undeclared = COUNT ;

/* k2 used before it is assigned to, flagged */
EXEC SQL
   SET descriptor 'ext_cur_name' COUNT = :k2;

EXEC SQL
   SET descriptor 'ext_cur_name' VALUE desc_id = :k1;

EXEC SQL
      UPDATE employee
         SET salary = :emp;

/* pno1 used before it is assigned to, flagged */
EXEC SQL
      SELECT SP.SNO
         INTO :val1
         FROM P
         WHERE SP.PNO = :pno1
   ;

some_global=val1; /* val1 was assigned to above */
}
```

OSPC supports the full SQL/2 standard. Most vendors have added their own extensions. The `-SQLV` option can be used to enable a particular vendor extension (Oracle, Ingres and Informix are currently support to varying degrees).

5.8 Lint checks

Lint is the name of a tool, written in the early days of Unix, designed to look for potential coding problems. Lint like problems might be regarded as constructs, which while conforming C, might be coding errors on the part of the programmer. The classic example is:

```
if (x = 3)
```

where the equality operator, `==`, operator was probably intended, not the assignment operator.

Switching on the `-Lint` option causes **OSPC** to flag lint like constructs.

Sources of information on what to regard as a lint like construct include: user requests, the Unix lint tool and various company coding standards.

5.8.1 Identifier usage

Externally visible identifiers should be declared in a header. As a consistency check this header should also be included by the file that defines the identifier. When enabled the lint option will cause any externally visible identifiers, not declared in an included header, to be flagged.

There are several possible situations where use of locally defined identifiers may be suspicious.

1 Value assigned to an object, which is not subsequently referenced. It may be possible to change the assignment to a void expression.

2 Defined identifier is not referenced. The definition can be removed. This check is also done for `static` definitions.

3 The object is only assigned to once. It may be possible to declare the object using the `const` qualifier and assign the value as an initialiser.

Host variables occurring within embedded SQL are also checked.

5.8.2 Use of headers

As a program evolves the headers files included at the top of a translation unit continue to grow. Header files that are included, but not referenced add to compilation times and complicate make files.

When enabled the lint option causes **mcc** to check the usage of each included header. Those whose contents are not referenced, by the main body of the C source, are flagged as such. Note that header `abc` may refer to an identifier in header `xyz`, but neither header be referred to from the including C file. If the `#include “xyz”` is removed the `#include “abc”` will also need to be removed.

5.9 Coding standards

Using the `-CODES` option tells **OSPC** to do coding standards related checks.

5.9.1 Code layout

While editing an existing program sections of code are sometimes moved around. It can be difficult to follow the nesting of conditionals in complicated functions and mistakes are made. The indentation of a statement is often a reliable guide to its intended nesting level. When enabled the lint option checks the indentation of statements and flags those that differ from previous statements in the same block.

Other layout issues include:

1 The use of braces, both in their presence and where they are placed. **mcc** attempts to flag inconsistent usage.

2 Multiple statements on the same line (multiple statements derived from the same macro invocation are not flagged).

3 Multiple declarations on the same line.

5.9.2 Implicit casts

The C standard does not require that two types be exactly the same. In many circumstances an implicit cast will be inserted. Switching on this check will cause ‘suspicious’ implicit casts to be flagged.

5.9.3 Loop checks

The C *for* statement is defined in terms of the *while* statement. Many coding standards mandate that the loop control variable only be modified within the loop header and not inside the compound statement. When this check is enabled **mcc** analyses loop headers looking for ‘suspicious’ constructs.

5.9.4 Appearance of a comment

Many coding standards require that major language constructs be preceded by a comment. When enabled this coding standard option flags any major construct that is not preceded by a comment.

5.10 Metrics

Software metrics do not give warnings about a specific source construct. Rather, they flag whole functions or expressions where a particular metric has exceeded some limit. The value of this limit being based on analysis of other code with a known problem history. Metrics are a statistical indicator of the likely hood of an error occurring in a portion of code. Another way to use metric information is to monitor how they change, for a given piece of software, over time.

5.10.1 Which metric?

Many different measures of software complexity have been proposed. Interestingly many correlate very highly to lines of code.

OSPC does not attempt to select among the competing metrics. Instead it focuses on producing the raw data from which most metrics can be calculated. This data is written to a `.met` file, if the `-METrics` option has been switched on. It is the job of other tools to calculate and display metric information. The source code of such a tool is provided in the software distributed with `OSPC`.

The contents of a `.met` file are described in the Reference Manual.

5.10.2 dispmet

The **dispmet** directory on the distribution tape contains the source of a program that takes a `.met` file and displays various metrics, using information from that file.

To create an executable from the source provided use the make file provided as part of the distribution.

5.11 make

It is intended that the **OSPC** be integrated into a companies standard development environment. The most common tool used to support the building of applications is **make**. Thus it is very important that existing make files be supported with the minimum of effort.

There are several methods of approaching this problem. We will discuss them here and outline their advantages and disadvantages. The final choice is left at the discretion of the user.

One problem that is common to all methods is where to put the .kic files. If they are placed in the same directory as the source and object code they may well cause excessive clutter. One possibility is to create a subdirectory and use the `-OUTPUTPath` option to redirect all output to this directory. If it is not intended to perform cross unit checking the `-CHECK` option can be used to switch off the generation of .kic files.

5.11.1 ccc

This shell script is provided as part of the standard distribution of **OSPC**. It's purpose is to replace the **cc** command in make files. The **ccc** command actually invokes **cc**, but before doing so it invokes **mcc**. Thus all files that are recompiled as a result of any changes made to them, or their dependencies are automatically rechecked as well as being recompiled. The **ccc** command will also invoke **mcl** to perform the cross unit checks if a link is requested.

At least one and perhaps two changes will need to be made to the make file to support the use of **ccc**.

1 Instead of setting the value of the `CC` macro on the command line, the line:

```
CC = cc
```

could be changed to:

```
CC = ccc
```

It is possible that this line does not already exist in the make file. In this case it can simply be added near the start of the file, with the other macro definitions. It is also necessary to ensure that `$(CC)` is used throughout the make file, not **cc**.

2 The lines:

```
.c.o :
       $(CC) $(CFLAGS) $<
```

should be added to the make file (note that the second line contains a tab character, not multiple space characters). Chances are that these lines do not already exist in the make file. They describe the default rules for creating a .o file from a .c file. Note that on some platforms the default rule for invoking **cc** also define some macros via the `-D` option.

3 If use is made of `ar` to build libraries the line:

```
AR = ar
```

has to be changed to:

```
AR = arr
```

It is possible that this line does not already exist in the make file. In this case it can simply be added near the start of the file, with the other macro definitions. Similarly within the body of the make file any occurrence of `ar` should be changed to `$(AR)`. This will cause libraries of .kic files to be created.

By having a make file that explicitly contains the rules for creating a .o file from a .c file the user at least knows what is going on. In some cases users may not want to change the value of the `CC` macro to be **ccc**. In this case **make** can be invoked with the command line:

make CC=ccc

Because it appears on the command line the assignment to `CC` will override any that may appear in the make file, or in the environment.

5.11.2 c99/c89/mcc

Using **mcc** instead of **ccc** has the advantage that it is not necessary to invoke the development compiler every time the code is checked. However, there is the danger that command line options may clash. The scripts **c99** and **c89** provide a **cc** compatible interface to **mcc** (they are named after the POSIX.2 tool for compiling C source). However, because **mcc** generates .kic files not .o files, additions will have to be made to the make file. Specifically:

1 A new suffix will have to be supported, add the line:

```
.SUFFIXES : .kic
```

to the make file. If a `.SUFFIXES` line already exists in the make file then add the .kic to the end of that line.

2 A rule for creating .kic files from .c files will need to be added:

```
.c.kic :
       c99 $<
```

The user has the choice of giving options to **mcc** here, via a make file variable or via the local options file.

The largest change that will have to be made to the make file is to change the names of the dependent object files from `xxx.o` to `xxx.kic`. This will invariably involve making a copy of the make file.

5.11.3 Scripts

The use of a make file reduces the amount of resources that need to be employed to check the software, after a change has been made. But did does not allow the user to control exactly which files are to be checked (**make** assumes that the application needs rebuilding). If only a few files are to be checked it may be simpler to use a script.

The following script file will cause all files ending in .c to be processed by **mcc**.

```
for file in *.c

  do

  mcc $file

done
```

or, under **csh**:

```
foreach f in (*.c)

  mcc $f

end
```

Here it is assumed that a local `.mccrc` file contains the required command line options.

This approach does not help much when it comes to cross unit checking. A via file, containing the names of all the files that are to be checked, will have to be created. This can be fed into **mcl** manually. Alternatively the command line:

```
mcl -o target.klc *.kic
```

might be used.

5.12 Other C compilers

Some C compilers have extended the language beyond that given in the C Standard. Use of language extensions ties an application to a given platform. However, there are two platforms that have sufficient market penetration to tempt users into using the extensions provided.

In the DOS world the architecture of the Intel 80x86 cpu provides for multiple ways of representing pointers. Extra keywords have been added into the language, by most compiler vendors, to support these representations.

The AT&T System V release 4 C compiler provides various extensions to the preprocessor. The most notable being `#assert` and the use of identifiers defined by this directive in `#if` directives. The SVR4 C compiler is available on all platforms running that release of the AT&T version of Unix.

The compiler in widespread use with probably the most extensions is the GNU compiler, GCC. Many of these extensions are used in systems headers on platforms that use GCC as the system compiler, for instance Linux.

The Open Systems Portability Checker supports the extensions available on both of these platforms. The `-EXtensions` option can be used to specify which platform extension needs to be supported while processing a particular translation unit.

Some compilers also support additional characters in identifiers. The `-IDStart` option can be used to specify which characters may occur at the start of an identifier. The `-IDFollow` option can be used to specify which characters may occur within an identifier. The dollar, '$', character is the most commonly supported additional character in identifiers.

5.13 c99

The POSIX.2 standard specifies **c99** as command used to compile C source programs. This name was chosen because it did not clash with the **cc** command. On most systems the, **c99** command, like **cc**, acts as a driver for the various components of the compilation and linking process. **OSPC** also contains a **c99** command, implemented as a shell script, that provides a front end onto **mcc** and **mcl**. The syntax follows that given in section A.1 of the POSIX.2 standard.

5.13.1 c99 options

-c	Compile but don't link

Compile (perform static checking on) the source file, leaving the .kic file, but don't attempt to link (cross unit check) it.

-g	Add symbolic information

mcc ignores this option, since the .kic files already contain full symbolic information. The flag is passed to the linker and assembler (if any .s files have been given.)

-s	Strip symbolic information

mcc currently ignores this option, but if specified the linker removes all non-essential information at link time.

-o <filename>	Specify output file

Send output to the named file, rather than `a.out`

-D <name><[=value]> Define macro

D <name> as if by a C language `#define` directive. If <=value> is omitted, a value of 1 is used. The `-D` option has lower precedence than the `-U` option. Hence if the same <name> is used both in a `-D` and a `-U` option on one command line, <name> will be undefined, regardless of the option ordering.

-E Preprocess to stdout

Copy C language source files to standard output, expanding any preprocessing directives.

-I <directory> Add include path

-l <library> Search the <library>

The Library named `lib<library>.a` will be linked into the new interpreter (**mce** option only) when linking (cross unit checking) is selected.

-L Change library search path

-O Optimize

Generate more efficient executable code (only really of use if the dynamic checker is to be used).

-U <name> Undefine macro

Remove any initial definition of the macro.

5.13.2 c99 file types supported

As well as acting as the interface for compiling C source programs, **c99** can also handle files with the following extensions:

file.c	A C language source file. This will be processed by **mcc** to produce `file.kic`.
file.s	An assembler file The file is passed to as, to the host assembler, producing `file.o`.
file.a	A host library of objects This library will be linked into the new interpreter (**mce** option only), and the users application when it is built.

file.kic	An intermediate code file. This will be passed to **mcl** if linking (cross unit checking) is specified.
file.klc	A linked intermediate code file. This will be passed to **mcl** if linking (cross unit checking) is specified.
file.o	An object file. Will be linked into the new interpreter (**mce** option only) if linking is performed.

5.14 Summary of options

Those options marked with a star* apply to using **mcc** in conjunction with **mce**, the dynamic checker.

Align <type>=<bytes>	Specify byte boundary for type
APIusage	Generate info on API usage
ARithrsh	Right shift (>>) to be arithmetic
ASsert <predicate>(name)	Predefine an assertion (System V.4)
BIGendian	Indicate order of bytes in a word
BITLohi	Bit-fields allocated lo-bit to hi-bit
BITOverlap	Can bit-fields share storage with previous field
BITSigned	Plain bit-fields to be signed
CHARConst	Format of multi-char character constants
CHARSet	Source character set (ascii, ebcdic)
CHECK	Perform syntax & semantic checks only
CHECKId <filename> **CHK <filename>**	Specify an ident checking filename
CODES <option>	Switch on the specified coding standard
CONDErr <filename> **CErr <filename>**	Specify an conditional error filename
COnfig <tag>=<filename> **CFG <tag>=<filename>**	Specify a configuration filename
D	#define
DETail	Display detailed help information
ECHO <text>	Echo text to standard output
ERRfile <filename>	Specify file containing error messages
ERRNumbers	Give OSPC internal error numbers with messages
EValorder <dirn>	Specify order of expression evaluation
EXtensions	Enable language extensions
FNAMEChar	Specify valid filename characters

FNAMELen	Maximum length of a filename
Forgetall <option>	Forget all arguments of option given so far
HEADers <filename>	Specify file containing valid system header files
HELPModify <modifiers>	Set modifiers for displayed help
HCEXcept*	Mark a system header as not being host compiled
HCI*	Host-compiled attribute inherited?
HCLIb*	Is the system library to be host compiled
HDRsuppress	Suppress warnings while processing system headers
HOSTComp* **<filename>** **HC <filename>**	Mark a header as being host-compiled
I <path>	Specify directory to search for `#include`
IDent	Check declarations against reserved identifier list
IDFollowchars	Characters that can occur in an identifier
IDStartchar	Characters that can start an identifier
INTErsperse	Intersperse listing with generated code
INTrep	Integer representation (1cmp, 2cmp, smag)
LIMits <identifier=value>	Set limits
LINT	Do Lint like checking
Listing	Generate a listing file
LOGfile <filename>	Specify a log file name
MAPfile	Generate a .map file
MAXErrors	Specify maximum number of errors
MAXWarnings	Specify maximum number of warnings
METrics	Generate software metric information
MISCId	Generate a platform specific id file
MODsign	Sign of result of signed division
NAMelength	Internal identifier character significance
NAMETrunc	Internal name truncation length
Nomsg <errnum>	Suppress a specific error number
OP <path> **OUTPUTPath <path>**	Specify output path for all output files
OPTimize*	Do code optimization
OSPCDir	Set directory, relative to source, in which to write .kic output
Output <filename>	Specify the output filename
PPlist	Produce preprocessed listing file
PRAGma <name>	Specify pragma supported by platform
PREInclude <filename>	Preinclude `<filename>`
PRINTFspec <specification>	Conversion specifiers supported by *printf*
PSId <filename>	Specify psid file

PTRScalar	Is conversion between pointer and same sized scalar ok
Quiet	Quiet mode
RAnge	Switch on pointer range checking
REFerences	Give standard reference on error messages
REMark	Comment option
SCANFspec <specifiers>	Conversion specifiers supported in *scanf*
SHEnd <string>	Execute string before termination
SHStart <string>	Execute string before compilation
Size <type>=<bits>	Specify type size in bits
SOurce <platform> **SRC <platform>**	Select source platform
SQL <level>	Enable the processing of embedded SQL
SQLV <vendor>	Specify the vendor dialect of SQL
SRCProfile <profile>	Select an additional source standard profile
STandard	Adhere rigidly to the ISO C standard
STACKDescend*	Ascending or descending system stack
STDHdr*	Use standard headers before system headers
STRuct <filename>	Read structure information from given file
SUMmary	Summarise the errors detected
SUppresslvl	Suppress messages below given level
SUWrap	Signed/unsigned conversions representable
TABwidth <width>	Set the number of spaces indented by the tab character
TArget <platform> **TGT <platform>**	Select target platform
TGTProfile <profile>	Select additional target standard profile
TRace	Trace reading of (config, include, input, profiles, options or memory)
Unsignedchar	Plain char to be unsigned
Verify	Verify order of evaluation of expressions
VIA <filename>	Specify control file to read further options from
XCasesig	Is case significant in external names
XNamelength **XL**	External name significance

Chapter 6

Conforming to an API

API’s (Application Program Interface) have become the method by which vendors define the software interface to their products. The product could be a piece of hardware, a third party library or even an operating system.

Users of applications often need to know which API’s an application relies on (for instance when purchasing hardware and software separately). Managers of development teams would probably like to know that only the defined API is being used and that the interface rules laid down in the specification are being followed (to reduce the likelihood of their product becoming tied to a particular version, or vendors implementation of an API).

An API specification defines the services that needs to be provided and how those services should be accessed. Here we are interested in checking how the services are accessed from the users source code.

6.1 What access methods might an API specify?

Information can be passed through a programming interface via function calls, or external variables. To hide implementation details symbolic names (usually macros, sometimes enumeration constants) are often used to represent special numeric values, *typedef*‘s are used to hide implementation types and structures are used to hold an aggregate of variables in one object.

To be of use the API must define more than the C syntax. It must define the properties of these names and the services they provide. For instance the external *xyz* represents a status flag set by the *abc* function and can take on any of the values given by the macros *A*, *B* or *C*.

The names of these functions, objects, macros and types are defined in one or more header files, to be included within the developers source code. The names of these header files and their known content is another part of the specification that can be checked.

6.2 Which API’s are used by an application?

Two things need to be done to answer this question:

1 Scan the applications source looking for all uses of external identifiers.

2 A database of API’s and the identifiers they define against which identifiers used in an application can be matched needs to be made available.

All references to external identifiers are matched against the contents of the API database, or other parts of the application (one unit may refer to an identifier defined in another unit, not an API). A match against an identifier contained in an API flags that API as being used (cases where different API's define the same identifier are rare and can usually be resolved by looking at the context, i.e., included headers and the use made of the identifier).

Identifiers that are not contained in another unit of the application, or the API database are regarded as referring to an unknown API (they could equally be referring to vendor extensions of a particular API).

6.2.1 Optional components

An API is sometimes broken down into core and optional components. For instance ODBC has a core and two optional levels; the real time portion of POSIX has 16 optional components. The availability of these components can be tested for using feature test macros within the application source code.

To be useful, any report of API usage has to list those optional components of an API that are used by the application.

6.3 Are the interface conventions obeyed?

It is no good making use of the facilities provided by an API if the interface specification is not followed. The whole purpose of an API is to isolate implementation details from the application. An applications that does not follow the specified interface is likely to have problems when using a new version of a library implementing that API, or the application is moved to a different platform.

So as well as finding out which identifiers are used, it is also necessary to check that they are used correctly.

6.4 Interface requirements specified in API's

API's specify a number of different requirements for correct usage. Commonly seen requirements include:

1 Types of functions and objects

2 Function arguments

  a) Symbolic names must be used

  b) Types of arguments must be compatible with a defined type

3 Function return values, or external object values

  a) Value has a properties, i.e., is positive, is negative

   b) Value may only be compared against symbolic names, or particular numeric literals

4 Feature test macros

   a) Used to check availability of optional constructs

5 Object types

   a) No requirement that the type be scalar

   b) Fields available in structs

   c) No requirement on layout, or ordering of fields

6 Identifier properties

   a) Symbolic name need not evaluate to a compile time constant

   b) Symbolic name must be implemented as a macro

   c) Name reserved for future use

7 Headers

   a) Must be included

   b) Inclusion reserves certain names

6.5 Function calls

6.5.1 Function arguments

The calling interface to some system service routines specifies symbolic values for one or more of the arguments. These symbolic values, implemented via macros, expanding up to implementation defined values. It is usually possible for a developer to find out the value of these symbols on a particular implementation and substitute the actual value, commonly used, for the symbolic name. Prior to the standardization of many API's it was common practice for numeric literals to be used.

Part of the API profile information specifies which functions must take symbolic names as particular arguments. **OSPC** uses this information to flag those calls that fail to obey the specified calling convention.

In:

```
#include <stdio.h>

fseek(file, 0, 0)  /* 3rd arg should be SEEK_SET */
```

the third argument should have been from the set *SEEK_SET, SEEK_CUR* or *SEEK_END*.

In some cases symbolic constants may be or'ed together to specify a combination of values. **OSPC** knows which symbols may be so joined and which may not.

```
#include <locale.h>

setlocale(LC_CTYPE | LC_TIME, str1);
setlocale(LC_CTYPE + LC_TIME, str2); /* may give different
                                        result to | operator */
setlocale((LC_CTYPE | LC_TIME) | LC_NUMERIC, str3);
```

Some API's have more complicated requirements. For instance the POSIX function *open* may take one of three values (*O_RDONLY, O_WRONY, O_RDWR*) combined with zero or more other values (*O_APPEND, O_NONBLOCK, O_NOCTTY, O_TRUNC, O_CREAT, O_EXCL*). An API checker must ensure that the argument is created using the correct boolean or of these macros.

The only operators that may be applied to symbolic names are bitwise and, *&&*, bitwise or, |, and the unary not ^ operators. The use of any other operator will be flagged. The ternary operator *?:* may be used, since it returns a single value.

Sometimes the numeric literal *0* (zero) is allowed, instead of a symbolic name.

Statically checking a variable passed as an argument is substantially more difficult. It requires full flow analysis to track the symbols assigned to that variable. The current release of **OSPC** does not perform such analysis in this context. It is assumed that the variable holds a correct value.

API's also define the types of the function argument. Provided the host compiler supports function prototypes the arguments given in calls to API functions will be checked at compile time. The C standard permits the passing of an argument of a different arithmetic type by requiring the compiler to insert a cast to the required type. For instance if *size_t* may have an *unsigned long* type, passing an argument of type *int* will work because of the implicit cast inserted by the compiler. Thus the developer does not have to worry about inserting casts to *size_t* for all appropriate arguments.

Passing an incorrect non-arithmetic type will cause the compiler to generate a compile time error (it is required to issue a diagnostic by the C standard). It is useful for an API checking tool to check that the arguments are compatible with the declared parameters, **OSPC** does this checking.

Some company coding standards require that arguments passed to functions are 'strongly compatible' with the argument type. That is the named types must matched. But this is a coding standards requirement, not an API requirement (because of the implicit casts inserted by the compiler).

6.6 Using the value of identifiers

Standards often restrict the values returned by calls to API routines to a range of values. This range of values may be expressed numerically, but is often expressed as a set of specific symbolic names. Also the values returned sometimes have properties, rather than specific values, i.e., is positive.

In the code fragment:

```
#include <stdlib.h>

int i;
char *c1, *c2;

/* ... */

while ((i == getchar()) != ‘a’)
    /* ... */ ;

if (strcmp(c1, c2) != 33)
    /* ... */ ;
```

the use of *getchar* in a comparison against an exact value is reasonable. This function returns a range of possible values and comparing for the appearance of one of these values is a common programming device. In the second case the exact comparison of *strcmp* is more suspect. The C standard specifies that this function returns zero, a negative value or a positive value. The number *33* is positive. However, there is no guarantee that all implementations of the *strcmp* function will choose this value to return in the positive case, or even that the same value will be returned from all calls. In this example **mcc** will flag the comparison against *33* and suggest an alternative (a relational comparison against zero).

The possible return values of functions and the range of values that may be assigned to objects contained in system headers forms part of an API profile.

In:

```
#include <errno.h>
#include <stdlib.h>
#include <time.h>

int i;
char *c1, *c2;
struct tm time_now;

/* ... */

errno = 99;  /* not a symbolic constant */
/* ... */

errno = EMTIMERS;   /* only defined in POSIX.4 */
/* ... */

if (time_now.is_dst == 1)
   /* ... */ ;
```

the first assignment to *errno* will be flagged (zero is the only defined numeric literal that may be assigned to this object). The second assignment is permitted, provided the optional POSIX.4 macro is available (in this case a feature test macro should have been used to check the availability).

API’s do not usually define many scalar types. A more common occurrence are *struct* types. The members of these types occasionally have restrictions placed on the values that they can take. For instance the tag *tm*, defined in `<time.h>`, contains a field that denotes the current setting of daylight savings time. This field can be a non zero positive value, zero or negative. The C standard does not define the positive or negative values used, rather it is the property of being positive or negative. So the final comparison, in the above example, is not testing for the positive property, rather it is testing for a particular positive value. This test may work under one implementation, but it is not portable. The comparison should be changed to greater than zero.

API functions may also return values. These values may represent individual values or particular properties, such as positiveness. For instance *printf* returns the number of characters printed or a negative value if an error occurred.

```
if (printf(“abc”) == 3) /* OK */
   ;

if (printf(“xyz”) == -1)
   ;
```

The first example is checking the number of characters written against the expected value, as allowed by the API. The second is assuming a particular value for the property of negativeness. One implementation may return *-1*, another *-2*, another an arbitrary negative value. The correct test would be:

```
if (printf(“xyz”) < 0)
   ;
```

here the relational operator is testing for the negative property.

A gray area of checking involves functions that return a limited range of values. For instance the *tm_sec* field of a *struct tm* may take on values between 0 and 61. Is the following code fragment relying on an implementation defined extension, or is it a coding error?

```
if (t.tm_sec > 61)
   ;
```

OSPC assumes that it is a coding error and flags this construct.

Like arguments, return values may sometimes be represented by symbolic names.

```
if (fflush(file_ptr) == EOF) /* OK */
   ;

if (fflush(file_ptr) == 1)
   ;
```

The second example is incorrect because it assumes a value for the symbol *EOF*.

Also relational operators may not be used in those cases where all the values returned by an API function are symbolic.

6.7 Optional constructs

6.7.1 Feature test macros

POSIX specifies some services as being optional. It also suggests a technique for using such optional services; the feature test macro. The idea is that a predefined macro can be used to check whether the services are available on a given platform.

```
#include <unistd.h>

#ifdef _POSIX_SAVED_IDS
   setuid(/* arguments */);
#else
   #error setuid may not be supported
#endif

#ifdef _POSIX_JOB_CONTROL
   func(WUNTRACED);
#else
   #error WUNTRACED not supported
#endif
```

In this example the *_POSIX_SAVED_IDS* feature test macro is used to protect the call to *setuid*. This macro will have been *#define*‘d in the header `<unistd.h>` if the service is available on the given platform. In some cases service routines take optional arguments, provided the option is supported. In the above case the *WUNTRACED* is only available if the *_POSIX_JOB_CONTROL* feature test macro is defined.

OSPC will check all uses of identifiers declared in standard headers to ensure that they are protected by feature test macros (where necessary). Those that are not protected will be flagged.

All forms of checking macros for existence are supported, as well as multiple conditions.

```
#include <unistd.h>

#if defined(_POSIX_SAVED_IDS) || defined(_POSIX_JOB_CONTROL)
   func()
#else
   #error what do we do now?
#endif
```

Note that only the macros actually used in calculated the truth of the *#if* expression are remembered. In the above case, if *_POSIX_SAVED_IDS* is defined the expression is true and the right hand side of the || operator is not evaluated. Thus any use of identifiers within the *#if* arm that depend on the *_POSIX_JOB_CONTROL* feature test macro being defined will be flagged, even though *_POSIX_JOB_CONTROL* may be defined. The user will have to split the test up into multiple *#if*‘s in this case.

The checking is based on simple existence and also applies to identifiers in the *#else* arm. The assumption being made that if a feature test macro is being used, it is probably being used correctly.

Optional constructs may be any identifier declared or defined by the API.

Developers that are unaware they are using optional constructs have set a future trap in the porting of their application. Users of packages also need to be aware of any optional constructs required by an applications when specifying hardware, or third party libraries.

6.8 Use of headers

6.8.1 Valid headers

A common method of accessing services provided as an extension to a standard is to use non-standard headers. **OSPC** checks the name of every system header included by the source code against a list of known, defined standard header names. Any unrecognized system header name is flagged.

```
#include <stdio.h>

#include <vendor_extension.h>

#include <X11/widgets/abc.h>

#include <X11/default/extension.h>

/* ... */
```

Here `stdio.h` is a permitted system header, while `vendor_extension.h` is not. In the case of X windows there are entire directories whose contents are considered reserved, irrespective of the name of the file. So the checkers life is more complicated.

If a header is not recognized **OSPC** will check its database of known header names. Associated with each header in this database is a list of platforms known to support that header. If a match is found a list of the platforms supporting the header is given. Thus providing some helpful information for the developer.

Headers are the means by which identifiers defined by in API may be made visible to the application. In some cases the header must be included because it contains information that cannot be obtained elsewhere (for instance the values chosen by the implementation for symbolic names). Sometimes it is possible for a developer to declare a subset of the API without including the header.

Headers are necessary if symbolic macros and types are referenced from the application source. For instance in the example involving *fseek* above the header `stdio.h` needs to be included so that the compiler can obtained the value of the macro *SEEK_SET* chosen by the implementation.

An example where an API header need not be include is the *strerror* function. It is ok to declare that function explicitly, rather than including the `string.h` header. Because its API specification only uses C predefined types, *char *strerror(int errnum)*. However, *memset* could not be so declared in the users source without including the `string.h` header (if the header is included why explicitly declare it anyway). This is because the declaration of *memset* needs a type from that header, *size_t*. The developer may declare *memset* with a particular predefined type instead of *size_t*, but that will only work on implementations where that type is used to represent *size_t*. (The C API specifies the type *void *memset(void *s, int c, size_t n)*).

6.8.2 Incorrect header contents

A problem that sometimes arises with API headers is that they do not accurately reflect the requirements contained in an API. Fortunately the most common problem, incorrectly specified argument arithmetic types, does not affect the performance of a checking tool. If, for instance, a vendors version of `string.h` declared the third argument of *memset* to take an *unsigned int* argument the interface is not broken from the applications point of view, provided *size_t* is also declared to have type *unsigned int*. The compiler vendor is at fault for not upgrading its headers to conform to the C standard (first published in 1989 by ANSI and as an ISO standard in 1992).

Other problems often seen include syntax violations (text after a *#endif* not included within comment delimiters for example) and incorrect numeric value for macros (floating point values inaccurate in the last digit).

6.9 Use of API defined types

An API may define types to allow implementations to adapt themselves to different hardware (usually different sized scalar types) or to combine together similar variables in one place (a structure).

API's rarely define the ordering of fields within a struct, although implementations are usually given liberty to add additional fields to structs. Applications that rely on ordering of fields or make use of implementation specific fields are going beyond the specification given in the API.

6.9.1 Struct fields

The amount of information returned, or passed to, API functions can sometimes be larger than will fit in a scalar type. In these cases standards define *struct* types to hold this information. They also specify the names of fields expected to be present in these structures. On some platforms the *struct*s defined in header files often include extra fields that are not defined in the standards (they are platform specific fields). Such behaviour is permitted by most standards and is a recognized method of providing extensions. But software that references these fields is making use of extensions and its conformance to standards is reduced. It also becomes tied to a particular platform, and has reduced portability. **OSPC** checks that all fields referenced in an object having the type of one of these standard *struct*s is defined as existing in the standards supported by the target platform.

A common example is the *dirent* structure defined in `<dirent.h>`. The POSIX.1 standard specifies it must include the member *d_name* with type *char []*. XPG goes further and specifies that the field *d_ino* must also be present. However, if we look at the documentation for the Sun 4 we find the following structure defined:

```
struct dirent {
              u_long    d_ino;
              short     d_reclen;
```

```
                short     d_namelen;
                u_long    d_name[MAXNAMELEN + 1];
                };
```

It is perfectly valid for the extra fields to be defined by a platform. But the extra fields are not portable.

So in the code fragment:

```
#include <dirent.h>

struct dirent cur_dir;

/* ... */

if (cur_dir.d_name[1] == ‘a’)  /* Conforming POSIX.1    */
   ;

if (cur_dir.d_ino == 0)        /* Conforming XPG        */
   ;

if (cur_dir.d_reclen == 4)     /* Valid on Sun 4 only   */
   ;
```

each access to *cur_dir* uses a successively less conforming field name.

The field names of structures defined by standards are held in the standards platform profile directory, along with the names of the fields defined by those standards. In the example given above XPG is a superset of POSIX.1. Thus its platform profile references POSIX.1 and simply includes that functionality that is supported in addition to POSIX.1.

6.9.2 struct initialisation

Initialization of struct objects, via an initialiser, is one example where an ordering of fields is implied. So the construct:

```
div_t local_var = {1, 2};
```

must be explicitly expanded out to (assuming the above assumed this order):

```
div_t local_var;

local_var.quot=1;
local_var.rem=2;
```

6.9.3 Type need not be scalar

An API occasionally leaves the specification of a particular type wide open. An example is the *fpos_t* typedef specified in the C standard, which simply states “... which is an object type capable of recording all the information needed to specify uniquely every position within a file.” On many systems this type is a scalar. So the code:

```
#include <stdio.h>

fpos_t *position1,
       *position2;

/* ... */

if (*position1 == *position2) /* comparison only legal if
```

```
                                        fpos_t has scalar type */
        /* ... */
```

works. But C does not allow the == operator to be applied to struct types. This code fragment would fail to compile on a platform that defined *fpos_t* to be a struct (in fact there is no portable way of comparing two objects of arbitrary type for equality).

6.10 Symbolic name need not be constant

API's use symbolic macro names to represent values that may vary between implementations. Developers sometimes assume that because macros are used the value will be a constant literal. This is sometimes not the case. For instance, of all the macros used to describe properties of the floating point representation, in the C standard, only one, *FLT_RADIX*, is required to be a constant expression. On many implementation the other macros are indeed constant expressions, but they are not required to be.

The code fragment:

```
#include <float.h>

int number[FLT_DIG];
```

relies on *FLT_DIG*, the number of decimal digits in a number that can be exactly represented in a float, being a constant expression. If it is an expression that must be evaluated, as above, at runtime the compiler will not be able to compile the application.

6.11 Declaration/Definition checking

Many standards define a large number of reserved names. These names may be reserved for several reasons:

- For use by an application. The names are functions, objects, macros and typedefs given in the standard.
- For use by an implementation. The names may be used by an implementation, for housekeeping purposes necessary to perform the required API functionality.
- For use in future versions of the standard. In this case names beginning or ending with certain sequences of characters are usually reserved.

Unwitting use of one of these reserved names now, may cause the contents of a source file to clash with a new version of an API library, an API on a different platform, or a future revision of the standard.

Within each standards profile there is a file containing a list of names reserved by that standard. On encountering a declaration of such a name, in the appropriate namespace, linkage and scope, a message is generated.

In the example:

```
#include <string.h>

static int strange;

int stream;

int __str_stuff;

/* ... */
```

the use of the header `string.h` causes all identifiers, starting with `str` and having file scope to be reserved. Objects with external linkage beginning with `str` are always reserved, since they might clash with a routine linked in from the system library. File scope identifiers beginning with __ (double underscore) are always reserved for use by the implementation.

The chapter on understanding the C standard (in the Understanding Standards manual) provides a fuller description of the conditions under which names may become reserved.

The `-CHECKId` option may be used to specify an ident checking filename. The named file is taken as the file containing a list of reserved identifiers. Multiple ident files can be specified, and identifiers will be checked against each of them in turn.

If the file cannot be opened, or is not in the correct format an error message is displayed and no checking for reserved identifiers is performed.

mcc prog -check PROFILE/standard/ansic/ident

6.11.1 Exceptions

Standards tend to reserve identifier without regard to the requirements of other standards. Thus from time to time, one standard will define an identifier that is reserved by another standard. X11 is a case in point. Here identifiers beginning with `_X` are recommended for internal use. This clashes with requirements contained in the C standard. To help solve this problem **OSPC** allows exception identifiers to be specified (this information is stored in the reserved identifier database for each standard).

For instance the X11 reserved identifier database specifies that all names beginning with `_X` should not be flagged.

The exceptions list is checked after an identifier matches against the list of reserved identifiers. If it also matches against the list of exceptions, no warning is given for that identifier's declaration or definition.

Like reserved identifiers, exceptions also require the context in which they are to be treated as exceptions.

6.11.2 Using #undef

It is possible for the user to 'win' back names that have been reserved by a standard. The `#undef` preprocessing directive undefines a previously defined macro. Thus it can be used to 'make safe' names that are required for use in the user's program. The `#undef` will remove any possibility of a macro defined in a system header causing a replacement in the users source (provided the `#undef` occurs after all the system headers have been included).

For example, any name beginning with `str` is reserved by the C standard:

```
#include <string.h>

#undef string1

static int string1;
```

The `#undef` protects the definition of `string1` against any macro of that name that may have been defined in `<string.h>`.

Note: Some standards define names that are so ‘strong’ that they cannot be ‘won’ back with `#undef`. This topic is discussed more fully in the manual explaining standards.

6.12 Reserved names

API’s often reserve specific names for future releases of the specification and to allow implementations to add additional names to headers.

The presence of these reserved names effectively constrains an application from defining identifiers with those names. An application containing such a definition could fail to compile on certain platforms, or with later versions of the API (because of duplicate or inconsistent definitions).

Some reserved names are easy to avoid by the application developer, for instance those starting with double underscore. Others might be considered more contentious. For instance all macros starting with *E* (capital E) are reserved by the C standard if the header `errno.h` is included, and all external identifiers starting with the three characters *str* are reserved in all cases by the C standard.

Experience has shown that applications often contain definitions of many identifiers whose names clash with those reserved by API’s. The definitions could be changed to use alternative names, but in many cases the effort involved is disproportional to the time and effort needed to modify existing code.

Insisting that all names defined by an application not clash with those reserved by the API’s used is impractical. It could mean having to edit tens of thousands of lines of code. Applications vendors might undertake not to add new names that are reserved and attempt to migrate away from any existing reserved names that are being used.

6.12.1 Platform specific identifier files

Most computing platforms provide services that go beyond those specified in standards. Software that makes use of these platform specific services had better make sure that it does so in a configurable manner.

One component of the platform profile information used by **OSPC** is a list of those macros, objects and functions that are specific to a given platform. Most platform profile information relates to the target platform. In the case of the platform specific identifier information it relates to the source platform. After all, the software already runs on the source platform and if it is making use of services specific to that platform it is unlikely to port directly to the target.

```
#include <memory.h>

/* ... */

bcopy(/* parameters */);

memccpy(/* parameters */);
```

In the first example `bcopy` is a method of copying between two objects on Sun platforms. This function does not exist in any standards, or even in a Sun header (it is default declared). It ought to be replaced by `memcpy` (which is not available on Sun running SunOS 4.1).

The function `memccpy` is again Sun specific. Its usage should be protected by a Sun feature test macro.

The types of warnings generated for platform specific identifiers will vary depending on whether the software is being check on the source platform or some other platform.

In the case of the source platform any header files that are include will contain declarations of the relevant identifiers. In the case of other platforms (perhaps the target) the included header files may not exist or, if they do, may not contain declarations of all of the identifiers that existed on the source platform. In the latter case it is likely that a syntax or semantic warning will be generated on any use of the source specific identifier. If it is a function call the user may have to wait until link time to see the message (unresolved external).

6.12.2 Identifiers specified to have type related properties

The C standard defines *errno* to "... expands to a modifiable lvalue that has type int ... It is unspecified whether errno is a macro or an identifier declared with external linkage." This is an example of an API defining properties of an interface rather than C syntax for its implementation. The POSIX standard says "... which is defined as extern int errno;" An implementation specification.

This kind of API object specification, using properties rather than C syntax is not very common.

Ideally an API checking tool would know about the different properties defined by an API and flag discrepancies. From the tools point of view such special cases are just that, special cases. In the example above it was reasoned that few applications rely solely on the C standard, most also include POSIX (or an API based on POSIX). So checks suggested by the specification given in the C standard is not carried out by **OSPC**.

6.13 Identifier specified by several API's

Sometimes a newer API will add functionality to an interface defined by an earlier API, or define what was previously undefined behaviour. For instance the C standard says that the *rename* function may be used to change the name of a file. But C has no concept of directory structure, so it does not include any specification for handling directories, it assumes a flat file system. POSIX defines a directory structure and adds to the specification to rename to describe how directories are to be handled.

It is not always possible to deduce the use being made of an API interface from static analysis of the source. **OSPC** takes the view that if an identifier from an API is referenced then that API is used, irrespective of the number of API's involved.

6.14 Can all referenced API's be detected?

No, they cannot. Consider the case of an API that only defines macros and types. Let us assume that information on this API is not available to **OSPC**. Who is to say that the included header, used to access the defined names, is not part of the application, rather than an API (**OSPC** makes the assumption that any header that appears within chevrons is a system header, so the use of such a header would be flagged as an unknown resource)? An example of a header that only contains macros and typedefs is `stddef.h`, from the C standard. An API rarely defines a single header containing macros and typedefs. Such headers are usually part of a larger collection of interrelated headers.

In the case of objects and functions their definition is contained in a library, not in the source making up an application. Such a library has the opportunity to modify an object and functions may access host specific information. So it is much harder for a user to duplicate the functionality in an application library.

Does it matter that use of an API may go undetected? Perhaps not. The developer has the option of taking the headers containing the macro and type definitions and making them part of the application source tree (if they are not available on a given platform). Of course the developer then has to take over responsibility for ensuring that the definitions are correct for each new platform.

In the ideal case the **OSPC** database contains information on all API's that an application uses.

6.15 Information output by an API checking tool

There are two types of files generated by **OSPC**:

1 .api files. This file contains information obtained from scanning a single source file.

2 .alg files. This file is created by collating all of the .api files corresponding to the source making up a complete application.

Being able to output a list of applicable API's relies on having a database of information about what each API contains. This is turn requires an API to be documented, which, unfortunately is not always the case (X11 being an example, where even the headers provided can vary between platforms, let alone the header contents).

6.15.1 Information summary

Once all of the source code used to build an application has been analyzed the information can be collated to give a summary of API usage. In the case of **OSPC** this summary, written to a .alg file, contains the information:

1 API

 a) All API's referenced

 b) Optional components referenced

2 Unknown resources used

 a) External identifiers referenced but not specified by a known API

 b) System headers included, but not specified by a known API

3 Violations of known API's

 a) Type of violation and number of occurrences

 b) Reserved identifiers used, and number of occurrences

6.15.2 Use of identifiers of unknown status

Once the entire application has been processed all referenced identifiers are known. Resolving these identifiers against those defined by the application and those defined by the known API's may leave some unaccounted for.

These unaccounted identifiers are assumed to belong to either an unknown API or extensions to a known API.

In the case of variables and macros there will be a declaration in one of the included headers. The name of the header may give clues to the status of the identifier.

Functions need not be declared prior to use. In this case the compiler will create a default declaration of *extern int f()*, where *f* is replaces by the name of the function. So there may not be a header name to refer to for guidance. Once again incorrectly written headers can confuse the analysis. It is not unknown for vendors to supply headers with some function declarations missing, even though code implementing that function is available in a library that can be linked against.

A checking tool can do no more than list identifiers whose status is unknown. This list may contain hints as to their likely status, for instance by giving the name of the header in which any declaration occurred.

6.16 Understanding the output messages

The following is a list of warnings that can be generated, along with an explanation of why are given and what might be done to stop them appearing. Switching on the `-REF` option will include any available API references to be given with the warning.

6.16.1 Arithmetic performed on object taking symbolic or discrete values

Unless an object takes a range of known values it makes no sense to perform arithmetic operations on it. The user is relying on information not required by the API specification.

```
errno++;
```

or,

```
errno -= 4;
```

both make little sense.

Of course it is possible to come up with API specifications that **APIdeduce** would flag as non-conforming (i.e., discrete values separated by known, constant, offsets). To date no such API specifications have been encountered.

6.16.2 Assigning an out of range value

As well as defining a list of symbolic values API's sometimes define a range of values that an object may take. Assigning a value outside this range (unless it is an allowed symbolic constant) will cause this warning to be given.

```
time.tm_min = 99;
```

6.16.3 Assigning symbol not given in standards profile

The list of symbolic constants that an API defines is known. A subset of this list includes symbols that may be assigned to objects Here a symbolic constant not on this sub-list is being assigned.

A very likely cause for this warning to occur making use of an extension to the API.

```
errno = E_OOOPS;
```

6.16.4 Assigning value not explicitly given in standards profile

An API may list a range of values, or properties, such as positiveness, that an object might take.

If the value being assigned either falls within the bounds of the minimum and maximum values possible, but is not within one of the explicit ranges. Or is one of the many values having a specified property, i.e., *1* is positive. Then this warning is given.

```
the_time.tm_isdst = 1;
```

6.16.5 Bad combination of symbolic constant: use X | Y | Z

The symbolic constants passed as an argument to an API routine have been combined in a form not supported by the API specification.

```
fseek(cur_file, 0, SEEK_SET + SEEK_CUR);
```

6.16.6 Bit-wise operations may not be performed on this symbolic object

API's often put restrictions on the values of symbolic constants, such that bitwise operations may be performed on them. This warning is given when no such restrictions have been specified by the API and bitwise operations are therefore not guaranteed to work.

```
flock_1.l_whence = (SEEK_SET | SEEK_CUR);
```

6.16.7 Comparison against a value not explicitly allowed in the API

API's sometimes specify a possible range of values that can occur. But without specifying that any of them may be explicitly mentioned. One such range are negative values. All numbers less than zero are negative. But an explicit test against one of them is not a test for the negative range.

This warning is given when a construct appear and the API specifies a range of values that might take occur, but without allowing any one of them may be compared against.

```
if (dup(1) == 22)
```

6.16.8 Comparison against symbol not given in standards profile

The list of symbolic constants that an API defines is known. Here a symbolic constant not on this list is being compared against. It is probably an extension to the API.

```
flock_1.l_whence = SEEK_BLK;
```

6.16.9 Comparison against value standards profile says cannot happen

As well as defining a list of symbolic values API's sometimes define a range of values that an object may take, and hence compared against. Comparing against a value outside this range (unless it is an allowed symbolic constant) will cause this warning to be given.

```
if (pipe(fildes) == 2)
```

In this example the call to *pipe* is only defined to return the values zero or one.

6.16.10 Dubious arithmetic performed on object taking symbolic or discrete values

Unless an object takes a range of known values it makes no sense to perform arithmetic operations on it. The user is relying on information not required by the API specification.

The reason for this warning is similar to the one given for the use of plus and minus assignment operators. Using a multiplication or other operator is even more unlikely to result in meaningful values.

```
errno *= 4;
```

6.16.11 Field ‘blah_blah’ of struct is not defined in the standard

APIs enumerate the members that a struct must contain. Vendors are usually given freedom to add additional members. Code that references a member added by a vendor may compile on one platform, but is unlikely to compile on another.

```
if (dir_info.d_off == 22)
```

The information provided by the vendor extension may not be available using the API specification only. The application vendor will have have to explain to their users why they have decided to make use of such an extension.

6.16.12 Header name not given in API

The name of the file between chevrons, < >, in a *#include* preprocessor directive is not a header name specified in any known API.

```
#include <vendorstuff.h>
```

Any header name enclosed in chevrons is treated as being a system header. headers nested within system headers need not be flagged since they are part of the implementation.

Sometimes developers use chevrons to delimit include filenames, rather than double quote characters. Such usage runs the risk of including an unexpected header on a different platform. Double quotes must be used for non-system header files.

6.16.13 Incorrect symbolic constant used: need one of {X|Y|Z}

The argument being passed to an API routine is not one of the correct symbolic values required. This warning often occurs because a particular argument requires more than one symbolic value and only one has been passed.

```
file_tag = open(“abc”, O_APPEND);
```

6.16.14 Initialiser assumes a specific ordering of fields

An explicit initialiser has been given for an object of *struct* or *union* type. Such an initialiser must depend on the ordering of members within the object. Relying on such an ordering is going beyond the API specification.

```
div_t local_var = {1, 2};
```

The solution is to explicitly assign an initial value to each member.

6.16.15 ‘memchr’ library function needs type from header file

The user is declaring a function that is also declared in a system header, but is not including that header. In most cases this usage is permitted by API specifications. But in the case that causes this warning the identifier being declared refers to a type that is declared in that, or another header. A declaration can be implicit or explicit:

1 A function is called with no visible declaration relying on the default declaration required to be created by the compiler.

2 The developer gives an explicit declaration in the source code, without including the appropriate header.

The correct way to gain access to the declaration is by including the header. It is very poor programming practice to rely on default declarations and in this case creating an explicit declaration in the source of the application does not have the desired effect.

```
void *memset(void *s, int c, size_t n);
```

Here *memset* takes a parameter of type *size_t*. The developer may know the type of *size_t* on the current platform, but cannot know its type on all platforms.

The solution is to include the appropriate system header and delete the declaration.

6.16.16 Macro ‘blah’ is not always a constant

Macros defined to hold numeric values are not always required to be constant literals. Although on many implementations they may have literal values.

```
char flt_digits[FLT_DIG];
```

6.16.17 Needs to be protected by the feature test macro _BLAH_

POSIX specifies that the availability of optional constructs may be tested for by using feature test macros. This warning is given when an optional construct is used without being protected by the appropriate feature test macro.

```
w_flags = WUNTRACED;
```

6.16.18 Nonsensical expression to assign to this object

This warning is similar to the one described above. The difference is that the object is being modified by a simple assignment not an operator assignment.

```
obj = SYM_CONST + 2;
```

6.16.19 Nonsensical expression to compare against this object

The expression being used to compare against an object contains operators that do not make sense in this context (given the API specification).

```
if (stat_1.st_mode == (S_IRWXG + 1))
```

6.16.20 Should assign a symbolic constant, not a literal

The purpose of a symbolic constant is to give a name to a number, removing the need for the user to know the actual value. Assigning a numeric literal is relying on information not given in the API specification.

```
errno = 3;
```

6.16.21 Should assign one or more symbolic values, not literals

Only symbolic values may be assigned to the given object, possible occurring in bitwise combinations.

```
stat_1.st_mode = 1;
```

6.16.22 Should compare against a symbolic constant, not a literal

The purpose of a symbolic constant is to give a name to a number, removing the need for the user to know the actual value. Comparing against a numeric literal is relying on information not given in the API specification.

```
if (errno == 9)
```

6.16.23 Should compare against one or more symbolic values, not literals

Only symbolic values may be compared against, possible occurring in bitwise combinations.

```
if (stat_1.st_mode == 1)
```

6.16.24 Should use symbolic constants (one of X | Y | Z)

The argument to an API routine is not one of the symbols given in the list in brackets. In a few commonly used routines programmers ‘know’ the value that ‘everybody’ uses and substitute the numeric literal rather than using the symbol. Implementors are not required to use ‘known’ values in their implementation.

```
fseek(cur_file, 0, 0);
```

6.16.25 Symbolic values should not be used in relational comparisons

The purpose of a symbolic value is that the actual numeric value used can vary between implementations. API specifications rarely define an ordering between the symbols. Use of a relational comparison relies on properties of an implementation that go beyond the API.

```
if (flock_1.l_whence > SEEK_SET)
```

6.16.26 ‘blah’ is reserved for future use

Many standards define a large number of reserved names. These names may be reserved for one of several reasons:

- For use by an application. These names are the functions, objects, macros and typedefs defined by an API.

- For use by an implementation. The names may be used by an implementation, for housekeeping purposes necessary to perform the required API functionality. They may also be extensions.

- For use in future versions of the API specification. In this case names beginning or ending with certain sequences of characters are usually reserved (POSIX reserves all identifiers starting with two underscore characters, or ending in *_t*).

Unwitting use of one of these reserved names now, may cause the contents of a source file to clash with a new version of an API library, an API on a different platform, or a future revision of the standard.

In the example:

```
#include <string.h>

static int strange;

int stream;

int __str_stuff;

/* ... */
```

the use of the header `string.h` causes all identifiers, starting with *str* and having file scope to be reserved. Objects with external linkage beginning with *str* are always reserved, since they might clash with a routine linked in from the system library. File scope identifiers beginning with __ (double underscore) are always reserved for use by the implementation.

The problem with this warning is that there tend to be lots of them. Editing all of the code to rename identifiers can be a time consuming task. It may also involve issuing new documentation or changing a defined internal user interface. Also in many cases the names used are unlikely to clash with names introduced by standards committees in the future.

This warning can also be given for API functions that are implicitly declared when a call is encountered. Because the names of API identifiers is usually reserved. To prevent this happening include the appropriate header file.

6.16.27 Using #undef

It is possible for the user to ‘win’ back names that have been reserved by a standard. The *#undef* preprocessing directive undefines a previously defined macro. Thus it can be used to ‘make safe’ names that are required for use in the user’s program. The *#undef* will remove any possibility of a macro defined in a system header causing a replacement in the users source (provided the *#undef* occurs after all the system headers have been included).

For example, any name beginning with *str* is reserved by the C standard:

```
#include <string.h>

#undef string1

static int string1;
```

The *#undef* protects the definition of *string1* against any macro of that name that may have been defined in `<string.h>`.

Note: Some standards define names that are so ‘strong’ that they cannot be ‘won’ back with *#undef*. This topic is discussed more fully in the manual explaining standards (part of the manual set that comes with the full **OSPC** distribution).

Chapter 7

OSPC cross unit checking

7.1 Introduction

The name of the **OSPC** tool that performs this role is **mcl**. The purpose of this tool is to check for interface inconsistencies across multiple translation units. The major cause of such inconsistencies are type incompatabilities. For instance an object declared as having `int` type in one translation unit and having `long` type in another. In practice most warnings refer to function declarations/definitions and arise as a result of forgetting to include the appropriate headers.

Those users familiar with the compiler/link/execute process will recognise this tool as being a sophisticated linker. Traditionally, linking is the process of joining two or more separately compiled source files, to create an executable program. The **OSPC** goes beyond this task and performs full interface checking.

Linkers, like compilers, have their own history and established way of doing things. **Mcl** has been designed with prior art in mind and can accommodate the behaviour (some would say deficiencies) of the well known linkers.

mcl performs three main functions:

1 It carries out type checking of declarations and definitions between translation units. This is its main checking role within the **OSPC**.

2 It reorganizes .kic files into a form suitable for execution; the traditional linkers job. This will only be of interest to those users who plan to execute the code generated by the **OSPC** at some point.

3 It provides a method of joining multiple .kic and klc files into one file, deleting and replacing translation units within a .klc file. This job might normally be carried out by a separate utility, a librarian, on other systems.

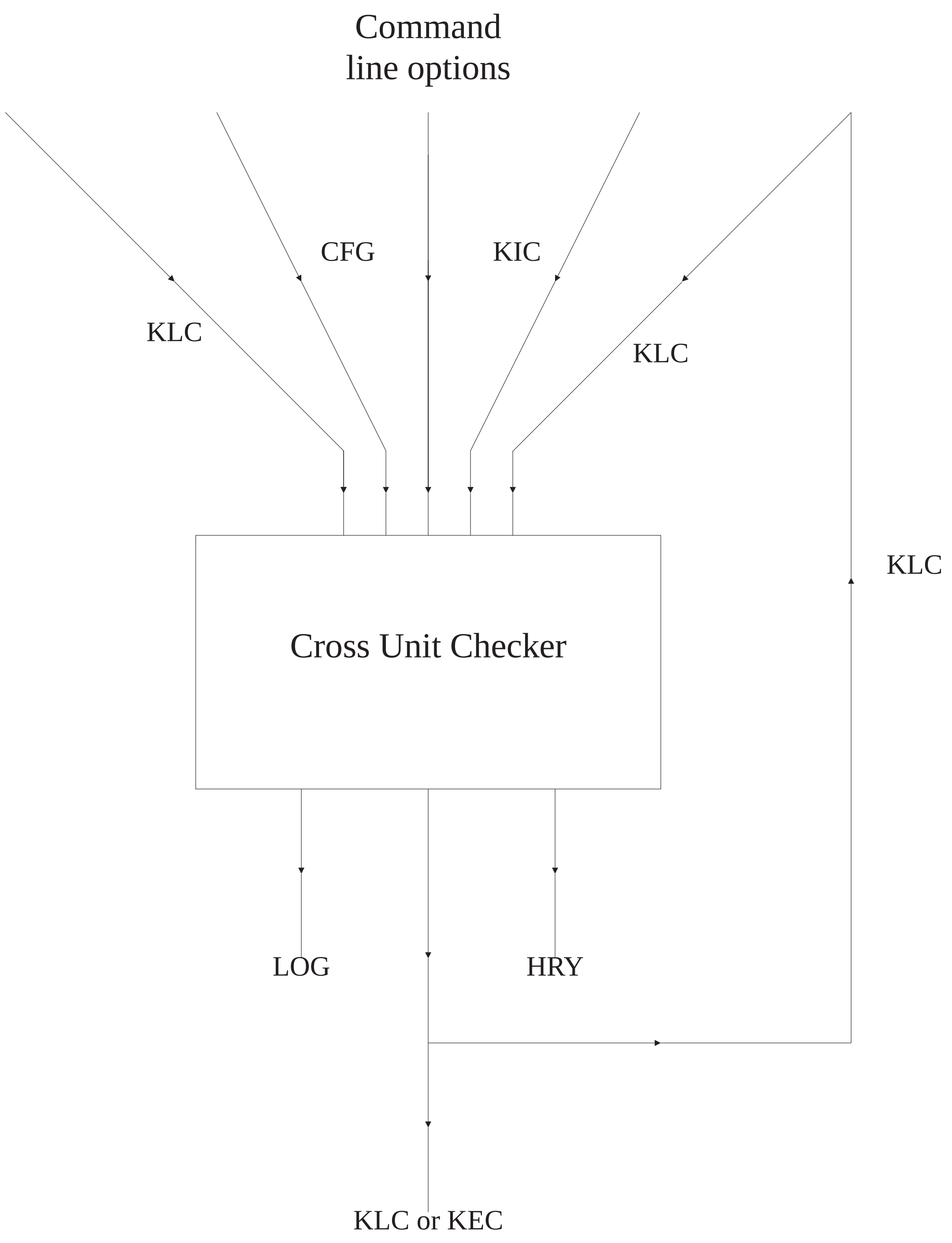
If you are planning to execute your programs, having them processed by **mcl** is an essential step. It is not possible to execute .kic files.

7.2 Using mcl

Mcl has the same user interface as **mcc**. Although the options available are different.

To process a single file type:

mcl abc



Linker Input and output files

This will cause **mcl** to read the file `abc.kic`, process it and generate a file called `abc.klc`.

To check more than one file type:

```
mcl abc def ghi
```

As many files as will fit on the command line may be given. The name of the first file is used to create the default full name of the klc file (in the above case the output file will be called `abc.klc`).

Mcl has a range of options that give the user control of the checking process. These options may be intermixed with the filenames:

```
mcl abc -ref def -V
```

The full list of options may be obtained by typing:

```
mcl -det
```

or simply (for a list of those options most commonly modified):

```
mcl
```

Because **mcl** does not work directly on the source code it is not possible to reference the point in the original file, that is the subject of a warning or error message. The best that can be done is to provide the name of the identifier involved and the names of the two files that contain the inconsistent declaration, definitions or usage.

When performing cross unit checks it is necessary to have a slightly different view point than that used for checking single source files. This need arises because the rules for type checking across translation unit in C are not the same as those within translation units. The cross translation unit type compatibility rules can be looked on as a super set of those used within a single translation unit. One possible simplification is for the developer, to only concern themselves with the source code compatibility rules and ignore the extra sophistication (some would say danger) of the additional freedom allowed at the cross unit level. But be warned, some declarations for which a warning message is expected may not in fact generate one. A description of the cross translation unit rules is given in the manual on understanding standards.

Inside a single source file ‘name equivalence’ is used. That is two types are the same if they have the same name; for scalar types this rules is extended somewhat. Across translation units type checking is by structure.

Because of its complexity, the full type checking procedure, as defined in the C standard, can sometimes consume a lot of machine time. For this reason ‘quick’ type checking was introduced. In this mode a 32 bit checksum is used. Each type is given a checksum. Type compatibility checking then simple involves comparing two check-sums. This technique, although much faster, has its drawbacks. It is possible for two different types to be given the same checksum value. However, this possibility is remote. What is more common is for two types that are compatible to be given different checksums (because C allows the same type to be written in different ways). Objects having these types are then flagged as being incompatable. Where possible the checksum has been created so as to minimise the possibility of two compatible types having different checksums.

The `-FUlltype` option controls which form of type checking is performed. Switching this option on causes the type checking to be as per the C standard.

mcl prog1 prog2 -fu-

If `file1.c` contains:

```
enum qw {a, b, c} x1;
enum ge {a1, b1, c1} x2;

struct gs {
           int mem1;
           long mem2;
          } gx;
int f1();
int f2();
```

and `file2.c` contains:

```
enum qw {c = 2, b = 1, a = 0}x1;
enum ge {a1, b1, ccc1} x2;

struct gs {
           int mem1;
           long mem2_2;
          } gx;
int f1(int);
int f2(char);
```

The warnings generated would depend on the setting of the `-FUlltype` option.

FUlltype+	x1 are compatible
	x2 are not compatible
	gx are not compatible
	f1 are compatible
	f2 are not compatible
FUlltype-	x1 are not compatible
	x2 are compatible
	gx are compatible
	f1 are compatible
	f2 are compatible

If 'quick' type checking is being used and there is some doubt as to the incompatability of the identifiers being flagged. It is recommended that full type checking be used. One advantage of using full type checking is that the types will be displayed in a readable form if incompatable types are found.

Note: Only those objects and functions that are referenced from the within the translation unit are written out. Thus if an object is declared with incompatable types in two units, but not referenced from one of those unit it will not be flagged. This can cause some surprise if the declarations are not changed and a new warning suddenly appears. The warning will have appeared because the object that was previously unreferenced is now referenced.

The C standard does not require that macros of the same name have the same bodies in different translation units. However, macros of the same name having different definitions is a common cause of problems. When the `-Body` option is switched on and more than one file is being checked **mcl** will check macros for consistency. This checking involves comparing the bodies, and in the case of function like macros the parameters, of macros with the same name. The checking rules used are the same as for multiple macro definitions within the same translation unit.

mcl part1 part2 -b-

If macro checking is not required and it is necessary to reduce the size of .kic files the `-MAcro` option can be used to stop macro definitions being written out. Switching this option off stops macro definitions being copied to the output file. If the option is on and there are multiple definitions of macros with the same name then only one of the definitions is copied (only those macros that are referenced are ever written out).

mcl part1 part2 -ma-

Some users are used to having their tools run silently. The `-Quiet` option provides just such a facility. Switching this option on causes output to standard output to stop. Output may be resumed by switching this option back off.

mcl part1 part2 -q+

If warning messages are being generated and the user is unsure which translation units are responsible the `-Verbose` option can be of use. Switching this option on cases **mcl** to display the name of every translation unit read in from a .kic or .klc file. If this name is a member of a library file it will be enclosed in *()*'s. When on this option also causes the full type of an object, or function to be displayed when incompatable types are found.

mcl part1 part2 -v-

If one or more files are linked a number of times it can be useful to review the interface check history. Every time **mcl** is executed it stores information in an audit trail. The `-ATV` option lists the audit trail contained in the input files being linked, together with the audit trail for this execution of **mcl** on the standard output.

mcl mylib -atv+

It can be tiresome always having to specify that the standard library has to be checked against. The `-Lib` option can be used to cause the .klc file (a prelinked form of the standard library) named in the configuration file always to be checked against the user's program. See the config file for details.

7.2.1 Character significance in identifiers

Historically linkers have not been as sophisticated as compilers in their handling of identifiers. Many linkers still only support upper case letters in identifiers and truncate after a relatively small number of significant characters. **mcl** is capable of mimicking this behaviour, more out of compatibility with prior art than a desire to hobble the user.

The `-XCOnvertcase` option controls the case folding of external names. Switching this option on causes external identifiers which only differ in their case to be treated as referring to the same object. Thus allowing **mcl** to subsequently act in a similar manner to a linker with such a restriction.

The C standard specifies that the case of external identifiers need not be considered significant.

If the following translation unit is processed by **mcl** using the `-XCOnvertcase+` option:

```
extern int a_long_name;
extern int A_long_name;
```

both declarations will be regarded as referring to the same object at link time.

If `file1.c` contains:

```
int a_long_name;
```

and `file2.c` contains:

```
int A_long_name;
```

Randomly selecting one of the definitions will not usually cause a problem. However, if both have different initial values assigned to them, then the final value of the object, prior to executing the first statement in `main`, is undefined.

The `-XNAMETrunc` option controls the significance of external names. This option may be used to specify the number of significant characters in an external identifier. Identifiers not significant after the given number of characters will be treated as referring to the same object.

The C standard specifies that only the first six characters in an external identifier need be considered significant.

If the following file is processed by **mcl** with the option `-XNAMETrunc 6`:

```
extern int a_long_name;
extern int a_long_day;
```

both declarations will be regarded as referring to the same object at link time.

The `-XCASESig` option controls the reporting of warnings based on the case of external names. Switching this option on causes a warning to be reported if two identifiers are the same except for their case, within their significant length. The identifiers are still treated as separate symbols. Thus **mcl** can flag identifiers that might cause problems with other linkers without having to behave like them.

As well as case significance the number of characters in an identifier can be significant. The `-XNAMELength` option controls the external name significance of identifiers. This option may be used to specify the number of significant characters in an external identifier. A warning will be given if identifiers aren't significant before the given number of characters.

The number of characters truncated to is stored in the klc file. Another invocation of **mcl** using truncation will cause the minimum of its truncation length and the previous truncation length to be stored in the .klc file. Thus when using **mcl** on a file previously truncated to six characters, with the option `-Name 8`, **mcl** will flag any clashes to within six characters.

7.3 Call/Definition checking

If a function is called in a unit that does not contain a declaration or definition for that identifier an implicit declaration is made. This implicit declaration has the equivalent effect to *extern f()*;

If `file3.c` contains:

```
/* ... */
int i;
long j;

f(i, j);
```

and `file4.c` contains:

```
/* ... */

long f(char c, int k)
{
/* ... */
}
```

then the command:

mcl file3 file4

will cause a warning to be generated to the effect that the function *f* is called with a different type to its definition. Definitions always take precedence over declarations. So if three files are being checked and one contains a call, one a declaration and one a definition the files containing the declaration/definition are checked first. If these are not compatible a warning is flagged. The definition will then be checked against the call.

One unit of source code may contain more than one call to the same function, and not contain a prototype. **mcc** will check that the type of each function call, deduced from the parameters passed, is compatible with every other call to that function in the same source file. If the same function is called with different parameter types it raises the question of which type **mcl** should check against. The solution adopted was to check against all of the call types. This approach ensures that no cross unit type mismatches go unflagged.

7.4 Using mcl as a librarian

The names of the linked files are stored in the klc file. Some operating systems support mixed case alphabetic characters in filenames, while others do not. The `-FOLD` option causes all characters in filenames to be forced to upper case before searching, or comparing with other filenames.

```
mcl part1 part2 -fold+
```

7.5 The call hierarchy

The `-HIerarchy` option causes a call tree to be generated. The hierarchy in question is a function call hierarchy. Starting at `main`, each called function is given. The diagram is organised to list all functions called by `main`, followed by all the functions that they call and so on.

Calls via pointers to functions are also given.

Note: This option can currently only be used in conjunction with linking to produce an executable (`-Exe` option).

```
mcl part1 part2 -hi+ -e+
```

Included with this call information is a list of functions assigned to each pointer, a list of uncalled functions and object definitions. At the end of the hierarchy diagram, along with the uncalled functions is a list of the unused file scope objects.

The name of the output file is created by appending the suffix .hry to the output filename (stripped of any .kec suffix).

7.6 Error reporting

Although they may be numerous the number of different causes for warnings, at cross unit check time, is small. **mcl** offers the standard range of reporting services provided by the common interface. There are no **mcl** specific error options.

7.7 Common warnings and their solution

Function declaration not compatible with definition. This problem is usually caused by a missing header file. If a function is defined using a prototype then calls from other translation units should also use the same type. If the header containing the function declaration, with or without a prototype, has not been included a call to that function will result in a default declaration (*extern f()*). It may also be possible that the header was included but does not include a declaration of the named function (older systems make use of the fact that a default declaration occurs to omit those declarations that are compatible with the implicit declaration).

‘x’ is defined more than once. This problem can arise because of multiple initialisations of the object in different translation units, perhaps even in a header that is included by different source files.

Object not defined. An object, with file scope, was referenced in one or more translation units making up the program, but no definition was found. This problem usually occurs because an object declared in a header has not been defined in any translation unit.

Options vary between files. Two or more of the files given to **mcl** were processed by **mcc** using different command line options.

7.8 Using make

Creating a runnable application involves several stages. Compiling the source was dealt with in the previous chapter. Once compiled the separate units need to be linked together. This linking process may be used to create libraries (for use by many applications), or it may produce an executable program. **cc** is the Unix tool that performs both compilation and linking. The replacement command **ccc** is also capable of carrying out both of these operation.

Thus if an existing make file makes use of **cc** to build an application then replacing it with **ccc** will ensure that the cross unit checks are carried out.

The Unix librarian, **ar**, can be used to manipulate libraries of .o files (the convention is that library files end in a .a). This also has an equivalent substitute, called **arr**. **arr** can be used to manipulate libraries of .klc files as well as .o files (it does this by invoking **mcl**).

There is a possibility that a make file has been written to invoke **ld**, the low level tool that actually does the work, rather than using **cc**.

7.9 Summary of options

Those options marked with a star* apply to using **mcl** in conjunction with **mce**, the dynamic checker. Foe a full description of these options see the Reference Manual.

ATP	Purge Audit Trail from file
ATV	View Audit Trail
Body	Do we check macro bodies between files
BUILDmce*	Build a new interpreter
CHKLIB*	Add checking routines for host compiled library
COnfig **CFG <filename>**	Read configuration from given file
Delete <filename>	Delete the given file from the input file
DETail	Display detailed help
ECHO <text>	Display the delimited text on the screen
ERRfile <filename>	Specify error file name
Exe*	Create an executable
FOLD	Fold filenames before comparing significance
Forgetall <option>	Forget all arguments of option given so far
FUlltype	Perform full cross translation unit checking
GLue*	Glue the .klc file to the new interpreter.
Graphics	Use graphics in the hierarchy report
HControl* **<filename>**	Specify the hierarchy control file
HIerarchy	Create a hierarchy file
HE	Entries in external hash table
HM	Entries in the macro hash table
HELPMod <modifiers>	Set modifiers for displayed help
HOSTinclude* **<filename>**	Link the file into the new interpreter
Keeptemp*	Don’t remove temporary files
Lib	Link in the standard libraries
LOGfile <filename>	Create a log file
MAcro	Macro checking between files
MAPFunc*	Remap a function
MAPUnit*	Remap the functions within a header
MCErts*	Set the path of the mce runtime system
Min	Use the least memory possible
MM	Memory the memory manager can use
MN	Number of nodes for the memory manager
Nomsg	Suppress a specific error number
OUTBuf **OB**	Set size of output buffer
OBJpath* **<path>**	Set path for finding host-compiled object files
OSPCDir	Set directory to find .kics and stub files
Output <filename>	Send output to the given file

OP **OUTPUTPath <path>**	Specify output path for all output files
Path <path>	Path for following files
Quiet	Stop displaying messages on standard output
REFerences	Display references to the standard
REMark	Treat delimited text as a comment
Replace <filename>	Replace the given file in the linked file
Search <filename>	Search path for .klc files
SUppresslvl	Suppress messages below given level
TRACECfg **TRC**	Trace Configuration being read in
TYpedepth	Maximum indentation for displaying types
Userlib <filename>	Add path to userlib in .klc file
Verbose	Talkative mcl
VIA <filename>	Take options from the given file
XCOnvertcase	Case fold external names
XCAsesig	Ignore case differences in externals
XNAMELength **XL**	External name significance
XNAMETrunc **XT**	External name truncation point
XTRact	Set record to delete from an executable

Chapter 8

Common Problems

This chapter contains a list of those questions and problems that commonly occur when using the **OSPC**.

Q How do I find out the default option settings in force when I run one of the component tools of the **OSPC**?

A Running that component without giving it any options (or giving the help option) will give a list of options and their current setting. The `-DET` option can be used to obtain a list of all available options and their current values

Note that this default setting is obtained by reading the `config` file and any local options file.

Q I made a local copy of the configuration file but it is not being read when I run its component tool.

A The `config` file is located by searching the `INFO/tool` directory. Use the `-CFG` option to cause the local copy to be used. Better still create a .rc file in the home directory.

Q I renamed one of the components of the **OSPC**. Now the `config` and `error` files are not being read.

A The appropriate `config` or `error` file is found by prefixing the name of the component tool to that suffix. Thus renaming **mcc** to **modc** will cause it to search for `INFO/modc/config`, `INFO/modc/error` and `INFO/modc/options.` So if a component tool is renamed its associated files will also need to be renamed.

Q The configuration and platform profile information is not being found.

A The directory containing this information is found by tracing back along the path that the tool being executed was found on. If the tool has been moved to another directory check that the directory `checkinfo` has also been modified.

Q System headers are not being found.

A Check that the correct include path is given in the `config` file or local options file.

Q Error numbers are being given instead of text messages. Why is this?

A If the component tool cannot locate and open the appropriate `error` file the error handling mechanism will display the error number. Check that the error files can be located and opened.

Q Sometimes error numbers are being given instead of text messages. The `error` file is being opened correctly. What is happening?

A The probable cause is missing lines from the `error` file. If the given error number cannot be found in the `error` file then the error handling mechanism will display the error number. Check that the error numbers being displayed do occur in the accessible error files.

Q The files being given to options are having a path prefix added to them.

A The `-OUTPUTPATH` and `-INPUTPATH` options sets the path prefix for output and input respectively. When encountered on the command line, or in a via file these options causes any filenames encountered in subsequent options to have this path prefixed to them. This behaviour can be switched off by using the `-Forgetall` option or by specifying a new path prefix.

Q Two names are flagged as being incompatible between two translation units and they look compatible.

A Make sure that **mcl** is not running in quick (option setting `-FUltype-`) mode. In C two declarations can look different and still be compatible. In quick mode a 32 bit checksum is used for compatibility checking. This use of this checksum approach can sometimes cause types that are compatible to be flagged as being incompatible. Check that the `-FUlltype` option is switched on.

Q When using ccc strange options are being passed to mcc.

A On some platform the **cc** command has default values for command line options built into it. **cc** itself invokes other programs to compile a source file. These internal values are passed to these other programs via command line options (unlike **OSPC**, **cc** does not read default options from configuration files). Because of the close coupling between **cc** and **ccc** these options are also passed to **mcc**. The solution is to provide an explicit rule in the makefile for building .o files from .c files.

Q A new platform profile has been created but the options it sets don't have the specified values.

A It is possible that an option is having its expected value overridden by a subsequent subprofile. Use `-TRACE profile` to see what values are being read in.

Q I am being told that the maximum number of users has been reached and that I cannot use the tool. But nobody else on the system is using **OSPC**.

A What has probably happened is that a running user aborted without telling the license manager that it had finished. This can happen if the `kill -9` command is used. The licensing system has a five minute timeout designed to handle this situation. Wait five minutes and try again. Aborting a process is quite a rare occurrence and not recommended, so this problem is unlikely to be seen frequently (using control C will not cause this problem).

Q Two objects, with the same name, in different files are incompatible, but **mcl** does not complain about them.

A The first thing to do is check that full type checking is enabled. There is a very small change that **mcl** will not flag incompatible declarations when in quick type checking mode. If the same answer is still being given check the C standard. Remember, across translation units the rules are different than for within a single unit.

Chapter 9

Collected Syntax

9.1 C Language

```
escape-sequence:
    simple-escape-sequence
    octal-escape-sequence
    hexadecimal-escape-sequence

simple-escape-sequence:
    one of
    \'  \"  \?  \\
    \a  \b  \f  \n  \r  \t  \v

octal-escape-sequence:
    \ octal-digit
    \ octal-digit octal-digit
    \ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
    \x hexadecimal-digit
    hexadecimal-escape-sequence hexadecimal-digit

token:
    keyword
    identifier
    constant
    string-literal
    operator
    punctuator

identifier:
    non-digit
    identifier non-digit
    identifier digit

non-digit: one of
    _  a  b  c  d  e  f  g  h  i  j  k  l  m
       n  o  p  q  r  s  t  u  v  w  x  y  z
       A  B  C  D  E  F  G  H  I  J  K  L  M
       N  O  P  Q  R  S  T  U  V  W  X  Y  Z
       $  (in non-strict C only)

digit: one of
    0  1  2  3  4  5  6  7  8  9

constant:
    floating-constant
    integer-constant
    enumeration-constant
    character-constant
```

floating-constant:
  fractional-constant exponent-partopt **floating-suffix**opt
  digit-sequence exponent-part floating-suffixopt

fractional-constant:
  digit-sequenceopt **. digit-sequence**
  digit-sequence .

exponent-part:
  e signopt **digit-sequence**
  E signopt **digit-sequence**

sign:
  +
  -

digit-sequence:
  digit
  digit-sequence digit

floating-suffix:
  f
  l
  F
  L

integer-constant:
  decimal-constant integer-suffixopt
  octal-constant integer-suffixopt
  hexadecimal-constant integer-suffixopt

decimal-constant:
  non-zero-digit
  decimal-constant digit

octal-constant:
  0
  octal-constant octal-digit

hexadecimal-constant:
  0x hexadecimal-digit
  0X hexadecimal-digit
  hexadecimal-constant hexadecimal-digit

non-zero-digit: one of
  1 2 3 4 5 6 7 8 9

octal-digit: one of
  0 1 2 3 4 5 6 7

hexadecimal-digit: one of
  0 1 2 3 4 5 6 7 8 9
  a b c d e f
  A B C D E F

integer-suffix:
  unsigned-suffix long-suffixopt
  long-suffix unsigned-suffixopt

unsigned-suffix:
  u
  U

long-suffix:
  l
  L

enumeration-constant:
  identifier

character-constant:
  ‘c-char-sequence’
  L ‘c-char-sequence’

c-char-sequence:
  c-char
  c-char-sequence c-char

c-char:
  any character in the source character set except the
  single-quote ‘, backslash \, or new-line character
  escape-sequence

string-literal:
  “s-char-sequence$_{\text{opt}}$**”**
  L “s-char-sequence$_{\text{opt}}$**”**

s-char-sequence:
  s-char
  s-char-sequence s-char

s-char:
  any character in the source character set except
  The double-quote “, backslash \, or new-line
  escape-sequence

operator: one of
  [ ] ( ) . - + - ~ ! / % ^ |
  ? : = , # sizeof
  ++ — & *
  < > = == != && ||
  ***= /= %= += -= <= >= &= ^= |=**
  ##

punctuator: one of
  [ ] ( ) { } * , : = ; ... #

preprocessing-file:
  group$_{\text{opt}}$

group:
  group-part
  group group-part

group-part:
  pp-tokens$_{\text{opt}}$ **new-line**
  if-section

control-line

if-section:
if-group elif-groups$_{\text{opt}}$ **else-group**$_{\text{opt}}$ **endif-line**

if-group:
# if constant-ex new-line group$_{\text{opt}}$
# ifdef identifier new-line group$_{\text{opt}}$
# ifndef identifier new-line group$_{\text{opt}}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
# elif constant-ex new-line group$_{\text{opt}}$

else-group:
# else new-line group$_{\text{opt}}$

endif-line:
# endif new-line

control-line:
# include pp-tokens new-line
# define identifier replacement-list new-line
# define ident lparen ident-list$_{\text{opt}}$ **) replace-list new-line**
# undef identifier new-line
# line pp-tokens new-line
# error pp-tokens$_{\text{opt}}$ **new-line**
# pragma pp-tokens$_{\text{opt}}$ **new-line**
# new-line

lparen:
the left-parenthesis character without preceding white-space

replacement-list:
pp-tokens$_{\text{opt}}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

preprocessing-token:
header-name (only within a #include directive)
identifier (no keyword distinction)
pp-number
character-constant
string-literal
operator
punctuator
each non-white-space character that cannot be one of the above

header-name:
h-char-sequence
“q-char-sequence”

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any character in the source character set except
the new-line character and >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any character in the source character set except
the new-line character and “

new-line:
the new-line character

pp-number:
digit
. digit
pp-number digit
pp-number nondigit
pp-number e sign
pp-number E sign
pp-number .

declaration:
declaration-specifiers init-declarator-list$_{opt}$**;**

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
init-declarator
init-declarator-list , init-declarator

init-declarator:
declarator
declarator = initializer

storage-class-specifier:
typedef
extern
static
auto
register

type-specifier:
void
char
short
int
long
float

```
    double
    signed
    unsigned
    struct-or-union-specifier
    enum-specifier
    typedef-name

struct-or-union-specifier:
    struct-or-union identifier_opt { struct-declaration-list }
    struct-or-union identifier

struct-or-union:
    struct
    union

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
    type-specifier
    type-qualifier specifier-qualifier-list

struct-declarator-list:
    struct-declarator
    struct-declarator-list , struct-declarator

struct-declarator:
    declarator
    declarator_opt: constant-expression

enum-specifier:
    enum identifier_opt { enumeration-list }
    enum identifier

enumeration-list:
    enumeration
    enumeration-list , enumeration

enumeration:
    enumeration-constant
    enumeration-constant = constant-expression

declarator:
    pointer_opt direct-declarator

direct-declarator:
    identifier
    ( declarator )
    direct-declarator [ constant-expression_opt ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-list_opt )

pointer:
    * type-qualifier-list_opt
    * type-qualifier-list_opt pointer
```

type-qualifier-list:
  type-qualifier
  type-qualifier-list type-qualifier

parameter-type-list:
  parameter-list
  parameter-list , ...

parameter-list:
  parameter-declaration
  parameter-list , parameter-declaration

parameter-declaration:
  declaration-specifiers declarator
  declaration-specifiers abstract-declarator$_{\text{opt}}$

identifier-list:
  identifier
  identifier-list , identifier

type-name:
  type-specifier-list abstract-declarator$_{\text{opt}}$

abstract-declarator:
  pointer
  pointer$_{\text{opt}}$ **direct-abstract-declarator**

direct-abstract-declarator:
  (abstract-declarator)
  direct-abstract-declarator$_{\text{opt}}$ **[constant-expression**$_{\text{opt}}$ **]**
  direct-abstract-declarator$_{\text{opt}}$ **(parameter-type-list**$_{\text{opt}}$ **)**

typedef-name:
  identifier

initializer:
  assignment-expression
  { initializer-list }
  { initializer-list , }

initializer-list:
  initializer
  initializer-list , initializer

translation-unit:
  external-definition
  translation-unit external-definition

external-definition:
  function-definition
  declaration

primary:
  identifier
  constant
  string-literal
  (expression)

postfix-ex:
primary-ex
postfix-ex [expression]
postfix-ex (argument-expression-list$_{opt}$ **)**
postfix-ex . identifier
postfix-ex - identifier
postfix-ex ++
postfix-ex —

argument-expression-list:
assignment-ex
argument-expression-list , assignment-ex

unary-ex:
postfix-ex
++ unary-ex
— unary-ex
unary-operator cast-ex
sizeof unary-ex
sizeof (type-name)

unary-operator: one of
& * + - ~ !

cast-ex:
unary-ex
(type-name) cast-ex

multiplicative-ex:
cast-ex
multiplicative-ex * cast-ex
multiplicative-ex / cast-ex
multiplicative-ex % cast-ex

shift-ex:
additive-ex
shift-ex < additive-ex
shift-ex > additive-ex

relational-ex:
shift-ex
relational-ex   shift-ex
relational-ex   shift-ex
relational-ex   shift-ex
relational-ex = shift-ex

equality-ex:
relational-ex
equality-ex == relational-ex
equality-ex != relational-ex

AND-ex:
equality-ex
AND-ex & equality-ex

exclusive-OR-ex:
AND-ex
exclusive-OR-ex ^ AND-ex

```
inclusive-OR-ex:
    exclusive-OR-ex
    inclusive-OR-ex | exclusive-OR-ex

logical-AND-ex:
    inclusive-OR-ex
    logical-AND-ex && inclusive-OR-ex

logical-OR-ex:
    logical-AND-ex
    logical-OR-ex || logical-AND-ex

conditional-ex:
    logical-OR-ex
    logical-OR-ex ? ex : conditional-ex

assignment-ex:
    conditional-ex
    unary-ex assignment-operator assignment-ex

assignment-operator: one of
    = *= /= %= += -= <= >= &= ^= |=

expression:
    assignment-ex
    expression , assignment-ex

constant-expression:
    conditional-expression

statement:
    labelled-statement
    compound-statement
    expression-statement
    jump-statement
    selection-statement
    iteration-statement

labelled-statement:
    identifier : statement
    case constant-ex : statement
    default : statement

compound-statement:
    { declaration-listopt statement-listopt}

declaration-list:
    declaration
    declaration-list declaration

statement-list:
    statement
    statement-list statement

expression-statement:
    expressionopt;

jump-statement:
    goto identifier ;
```

continue ;
break ;
return expression$_{opt}$**;**

selection-statement:
if (expression) statement
if (expression) statement else statement
switch (expression) statement

iteration-statement:
while (expression) statement
do statement while (expression) ;
for (expression$_{opt}$**; expression**$_{opt}$**; expression**$_{opt}$**) statement**

9.2 Precedence of operators

primary expressions

16	literals names	simple tokens
16	a[i]	subscripting
16	f()	function call
16	.	direct selection
16	->	indirect selection

unary expressions

15	++ —	postfix increment/decrement
14	++ —	prefix increment/decrement
14	sizeof	size
14	(type-name)	cast
14	~	bitwise not
14	!	logical not
14	-	arithmetic negation
14	&	adress of
14	*	contents of

binary operators

13L	* / %	multiplicative
12L	+ -	additive
11L	<< >>	shift
10L	=	inequality
9L	= = !=	equality
8L	&	bitwise and
7L	^	bitwise xor
6L	||	bitwise or
5L	&&	logical and
4L	|	logical or
3R	?:	conditional
2R	= += -= *= /= %= <<= >>= &= ^= |=	assignment
1L	,	comma

L, indicates left associative operators; R, right associative operators.

INDEX